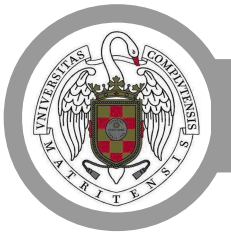

UNIVERSIDAD COMPLUTENSE DE MADRID
Facultad de Informática

Sistemas Informáticos 2009-2010

Knuthians

Videojuego Educativo para la Enseñanza y el Aprendizaje de las Gramáticas de Atributos.

Autores:
RAFAEL FERNÁNDEZ LÓPEZ
DANIEL RODRÍGUEZ CEREZO
ÁNGEL VALERO PICAZO

Director:
JOSÉ LUIS SIERRA RODRÍGUEZ

Knuthians: Videojuego Educativo para la Enseñanza y el Aprendizaje de las Gramáticas de Atributos.

Rafael Fernández López
Daniel Rodríguez Cerezo
Ángel Valero Picazo

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Rafael Fernández López Daniel Rodríguez Cerezo Ángel Valero Picazo

Resumen

En este proyecto se desarrolla un juego educativo llamado Knuthians. Este juego sirve para facilitar el aprendizaje de la asignatura "Procesadores del Lenguaje", ayudando al estudiante a comprender el mecanismo de propagación de los atributos en los árboles sintácticos, teniendo que establecer el flujo en el que los atributos (tanto sintetizados como heredados) fluyen a lo largo de estos árboles. En el proyecto también se ha desarrollado una herramienta que permite al profesor configurar el videojuego, planteando distintos ejercicios de propagación de atributos.

Palabras clave: XML, Videojuego educativo, Gramáticas de atributos, Árbol sintáctico, e-Learning, Motor de videojuego, Procesador de lenguaje.

Abstract

This project is an educational game called Knuthians. This game will be used to facilitate learning of the subject "Language Processors", helping students to understand the mechanism of propagation of the attributes in the syntactic tree, having to set the flux in which the attributes (both synthesized and inherited) flow along these trees. Along with this project has also been developed a tool that allows the teacher to set up the game, raising different exercises.

Keywords: XML, Educational videogame, Attribute grammars, Syntax tree, e-Learning, Game engine, Language processor.

Índice general

Capítulo 1

Prólogo

1.1. Introducción

El tema abordado en este proyecto de Sistemas Informáticos es el desarrollo de un videojuego educativo para enseñar cómo se resuelven las dependencias entre los atributos de un árbol sintáctico, producido por el análisis de una sentencia por una gramática de atributos. Las gramáticas de atributos, expuestas por Knuth a finales de los sesenta [D.E. Knuth 1968], son, hoy en día, un formalismo ampliamente empleado para la especificación del análisis y procesamiento de lenguajes. Actualmente se sigue enseñando este formalismo en las aulas de la Facultad de Informática de la Universidad Complutense de Madrid, más concretamente en las clases de *Procesadores de lenguajes*. Pero este formalismo no es fácilmente entendible. Por ello, se propuso convertir los árboles de análisis producidos por estas gramáticas en un juego que enseñase al alumno su funcionamiento.

1.2. Objetivos del proyecto

El principal objetivo de este proyecto de Sistemas Informáticos es la creación de un videojuego que permita enseñar el funcionamiento interno de las gramáticas de atributos. Más concretamente el videojuego debe permitir:

- El videojuego debe presentar el árbol sintáctico a resolver como un laberinto, y los atributos como objetos con los que el usuario podrá interactuar.

- La posibilidad de cargar distintos ejercicios.

Además, para facilitar la labor del docente a la hora de crear nuevos ejercicios para sus alumnos, se decidió crear un editor para construir los árboles sintácticos que posteriormente se cargarán en el videojuego para construir el laberinto.

1.3. Estructura del documento

La estructura de la memoria es la siguiente:

- En la segunda sección se realiza una introducción a los conceptos y tecnologías más relevantes en el proyecto, así como una breve introducción a las herramientas existentes y las utilizadas en el proyecto.
- En la tercera sección se describe el videojuego Knuthians, presentando sus distintas pantallas y las distintas funcionalidades dentro del videojuego. También se presenta la herramienta de creación de ejercicios.
- En la cuarta sección se detalla la arquitectura e implementación del videojuego Knuthians y de la herramienta de creación de ejercicios.
- Por último, en la quinta sección se presentan las principales conclusiones obtenidas tras la finalización del proyecto, así como posibles extensiones y trabajos futuros.

Capítulo 2

Revisión de conceptos y tecnologías

2.1. Introducción

En este capítulo se presenta una introducción a los aspectos conceptuales y tecnológicos más importantes que tienen relación directa con el proyecto, así como una introducción a las herramientas software más importantes que se han utilizado en el mismo.

Knuthians es un videojuego educativo por lo que en primer lugar se hará una breve explicación sobre este mundo, profundizando en otros proyectos basados en videojuegos educativos desarrollados en la facultad. Después se explicará qué es exactamente un motor de videojuegos, detallando sus partes y componentes, para finalmente comentar las características de distintos motores de videojuegos actuales y compararlos. Seguidamente, y dado que estos juegan un papel importante en el desarrollo de Knuthians, se hará una breve introducción a los lenguajes de marcado, haciendo especial hincapié en XML. Para finalizar esta sección se explicará el paradigma de las gramáticas de atributos, íntimamente ligado con el objetivo pedagógico de este juego.

2.2. Desarrollo de videojuegos educativos

Knuthians es un videojuego que pretende enseñar el funcionamiento de las gramáticas de atributos. Este hecho clasifica a Knuthians dentro de la

categoría de los videojuegos educativos.

Durante años una inmensa colección de programas educativos han visto la luz pero, desgraciadamente, en la mayoría de los casos la interacción con el usuario es muy limitada, haciendo que la motivación inicial por el aprendizaje se diluya en la constante visión de contenidos educativos poco interactivos. Para mejorar la motivación, en los últimos años han empezado a surgir un nuevo tipo de aprendizaje por computadora utilizando videojuegos, en lo que se ha venido a llamar aprendizaje basado en juegos o "juegos serios". En esta sección abordamos brevemente estos aspectos.

Comenzamos la sección dando un visión general de la informática educativa y e-Learning. Seguidamente profundizaremos en el concepto de los videojuegos educativos, para describir finalmente unos proyectos de ejemplo desarrollados en la facultad que ilustran este tipo de videojuegos.

2.2.1. Informática educativa y e-Learning

A lo largo de los años, educadores de los distintos niveles de educación, empresas y defensa han usado la informática de diferentes formas para ayudar y mejorar la enseñanza. De esta forma, se han acuñado distintos términos (aprendizaje basado en computador, enseñanza basada en computador, aprendizaje soportado por la tecnología, etc.) que propugnan diferentes maneras de utilizar la informática y las tecnologías de la información y las comunicaciones en los procesos de enseñanza y aprendizaje. Actualmente, todas estas tendencias confluyen en el concepto genérico de e-Learning [Nicholson Paul 2007].

En el contexto de la educación en general, el uso del término "e-Learning" ha tenido grandes connotaciones que abarcan un gran rango de prácticas, tecnologías y teorías. No está centrado únicamente en el contexto online, sino que incluye un gran abanico de plataformas de aprendizaje basadas en computadores y distintos medios de difusión, géneros y formatos como multimedia, programas educativos, simuladores, juegos y el uso de las nuevas tecnologías móviles a través de todas las disciplinas de la enseñanza.

De esta forma, y a modo de resumen, el e-Learning se puede definir de forma general como:*Enseñanza a distancia caracterizada por una separación espacio/temporal entre profesorado y alumnado (sin excluir encuentros físicos puntuales), entre los que predomina una comunicación de doble vía asíncrona, donde se usa preferentemente Internet como medio de comunicación y de distribución del conocimiento, de tal manera que el alumno es el centro de una*

formación independiente y flexible, al tener que gestionar su propio aprendizaje, generalmente con ayuda de tutores externos[Garcia Peñalvo 2005].

2.2.2. Videojuegos para educar

Uno de los campos más activos en e-Learning es el de los videojuegos educativos [Iván Martínez-Ortiz 2006]. Usar juegos (no solo videojuegos) para enseñar no es una idea nueva. A lo largo de la historia de la educación se ha mantenido la idea de que algo que es divertido es más difícil de olvidar. Con esta simple premisa los educadores y pedagogos han desarrollado algunos juegos para enseñar diversas cosas, sobre todo centrados para la enseñanza a temprana edad. Pero estos juegos tienen una limitación de medios que no los hace muy atractivos. Gracias a la tecnología estas limitaciones son superadas, pudiendo desarrollar videojuegos con los que poder educar.

Los videojuegos permiten la inmersión del estudiante en un mundo fielmente recreado con un bajo coste por estudiante. También permiten muchos enfoques educacionales, como competir y/o colaborar dentro del juego, o bien simulando compañeros virtuales con un bajo coste, o bien jugando en red con otros compañeros. Además los videojuegos son divertidos y consiguen atraer la atención de los estudiantes. Estas consideraciones se derivan de cuatro aspectos observados en y alrededor de los videojuegos modernos [Garris, R. 2002]:

- *Los videojuegos son divertidos.* Y no solo para los niños y los adolescentes. La edad media de los jugadores de videojuegos está alrededor de los 30 años, y más de 43 % de los jugadores son mujeres. Esto sugiere que los videojuegos educativos pueden llamar la atención a un amplio espectro de la población y que no son sólo aplicables a los estudiantes de primaria y secundaria, sino que se puede incorporar a trabajadores y estudiantes universitarios.

- *Los videojuegos son inmersivos.* Los videojuegos ofrecen la posibilidad de meterse en la piel de un personaje y recorrer el mundo creado por el diseñador e interactuar con él mediante distintas interfaces. En definitiva, los videojuegos permiten transferir la identidad del jugador dentro de la realidad creada por el diseñador. En los detalles psicológicos de esta inmersión y la transferencia de identidad se elaboran los videojuegos que pueden servir para la educación constructivista y ayudar en el proceso de aprendizaje de contenidos.

- *Los videojuegos estimulan la cooperación/competitividad.* Existen videojuegos que permiten jugar con o contra otros jugadores (reales o artificiales) para alcanzar los objetivos. Desde un punto de vista pedagógico esto es beneficioso porque ayuda al aprendizaje colaborativo sin ser necesario la actuación de compañeros reales. Además la competitividad ayuda a motivar a los estudiantes a aprender e intentar ser mejores que el rival.

- *Los videojuegos estimulan la creación de comunidades de practica.* Aunque algunos videojuegos no permitan un modo multijugador, es un hecho que algunos usuarios de determinados videojuegos forman comunidades on-line donde comparten experiencias, trucos, atajos y guías sobre el videojuego que otros usuarios pueden consultar. Desde un punto de vista pedagógico esto sugiere un potencial para la discusión de los conceptos que los videojuegos educativos pretenden enseñar, idealmente dirigida por un instructor experto.

Independientemente de las ventajas que poseen los videojuegos educativos, estos no servirán de nada si no son capaces de integrarse en la actual red e-Learning. Actualmente las prácticas más usuales para incluir videojuegos educativos son: (i)incluidos en objetos de aprendizaje [Stephen Downes 2001] como pequeños juegos centrados en áreas muy específicas de un tema, (ii) o usados como método de evaluación del progreso del alumno.

Por todo lo expuesto, los videojuegos educativos se están convirtiendo poco a poco en una nueva y poderosa herramienta al servicio de pedagogos y tutores.

2.2.3. Ejemplos de enfoques al desarrollo de videojuegos educativos.

Para finalizar esta sección se presentan dos ejemplos de enfoques al desarrollo de videojuegos educativos previamente formulados en la Facultad de Informática de la UCM: e-Adventure y JV2M.

e-Adventure

e-Adventure es una plataforma que aspira a facilitar la integración de juegos educativos y simulaciones basadas en juego en procesos educativos

en general y Entornos de Aprendizaje Virtuales (VLE) en particular [Daniel Burgos 2007]. Se centra en tres aspectos fundamentales:

- Reducción de los costes de desarrollo para videojuegos educativos.
- Incorporación de características educativas específicas en herramientas de desarrollo de juegos.
- Integración de los juegos resultantes con cursos existentes en Entornos de Aprendizaje Virtuales.

Esta plataforma nos presenta un editor de juegos con una interfaz amable y de fácil acceso para facilitar y reducir los costes de creación de videojuegos educativos. De esta forma, e-Adventure pretende acercar el desarrollo de videojuegos educativos a educadores y tutores. También incluye un sistema para que, automáticamente, el motor del juego registre información del grado de aprendizaje del usuario y la reporte al tutor. Además, el editor permite exportar los videojuegos creados como objetos de aprendizaje según los estándares más usuales. A fin de permitir lograr estas ventajas, e-Adventure restringe el género de los videojuegos creados a cierto tipo de aventuras gráficas [Esther del Moral 1996]. Así mismo, propugna la creación de juegos de muy bajo coste mediante la integración directa de fotografías digitales para configurar las escenas del juego. La Figura 2.1 [1] muestra algunos ejemplos de juegos creados con esta herramienta.

[1] Imágenes extraidas de http://e-adventure.e-ucm.es

Juego del Hematocrito.

Protocolo de Incendios.

Figura 2.1: Imágenes de juegos creados con e-Adventure.

JV^2M

JV^2M es un sistema educativo basado en videojuegos que pretende enseñar el funcionamiento interno de la máquina virtual de java [Gómez Martín Marco Antonio 2007]. Para ello presenta una ciudad donde los edificios y los objetos dentro de estos representan cada una de las partes de la maquina virtual, y el usuario deberá ejecutar un programa Java usando los elementos y edificios que encuentre en esa ciudad. La figura 2.2 muestra algunas capturas de este juego.

Figura 2.2: Capturas de JV^2M [Gómez Martín Marco Antonio 2007].

En JV^2M, el usuario maneja un avatar que se mueve dentro de un mundo que representa la maquina virtual de Java. Un aspecto bastante novedoso que presenta este videojuego es la inclusión de un tutor inteligente integrado dentro del videojuego, un sistema inteligente representado como un acom-

pañante del avatar al que el usuario puede hacer preguntas o incluso resolver el problema propuesto. Otro aspecto importante de este proyecto es la arquitectura interna del juego, ya que separa de forma muy clara los contenidos a enseñar con la forma de presentarlos.

Al contrario que e-Adventure, los juegos creados con JV^2M tienen una calidad más profesional. No obstante, el ámbito está más restringido: en concreto, resolución de problemas que involucran la compilación de programas Java en bytecode. Para tal fin, JV^2M puede configurarse a partir de descripciones de dichos problemas, junto con el resto de información necesaria para asistir al proceso de resolución por parte del alumno [Gómez Martín Marco Antonio 2007]. En Knuthians se va a seguir, de esta forma, un enfoque similar, permitiendo la configuración del juego a partir de problemas de propagación de atributos en árboles de análisis sintáctico.

2.3. Motores de Videojuegos

La construcción de un videojuego desde cero es un proceso inmensamente costoso. Es por ello que, para facilitarlo, se han creado marcos de trabajo específicos para la construcción de este tipo de programas: los motores de videojuegos. En esta sección se proporcionarán unas nociones básicas sobre los conceptos más importantes que se engloban en dichos motores. Así mismo, se expondrán también algunos de los motores de libre distribución que hoy en día son más conocidos y más usados, analizando, para cada uno de ellos, sus aspectos más importantes: el lenguaje en el que se programan los juegos, gráficos, sonido, física, efectos, plataformas que soportan dichos motores, etc. Al final de la presentación de cada motor se adjuntan algunos ejemplos gráficos para poder apreciar el potencial de cada uno de ellos.

2.3.1. Conceptos Básicos

Un motor de videojuego es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego [Ricardo Tejeiro 2003]. La analogía con el motor de un automóvil es ilustrativa: el motor debajo del capó no es visible, pero le da su funcionalidad al automóvil, que es la de transportar. La misma analogía permite explicar algunos de los aspectos que generalmente maneja un motor de juego: por ejemplo, las texturas y los modelos 3D corresponderían

a la carrocería, pintura y exteriores. Del mismo modo que carrocería, pintura y exteriores de un automóvil no son funcionales sin un motor al que son añadidos, los gráficos y los guiones del juego no funcionan sin un motor de juego.

A continuación se explican los principales conceptos referentes a un motor de videojuego.

- Assets (Activos): Son los modelos, animaciones, sonidos, IA, leyes físicas, etc. que forman el juego en sí. De esta forma, el código hace funcionar los assets.

- Application Programming Interface (Interfaz de Programación de Aplicaciones): Consiste en un sistema de rutinas, protocolos y herramientas para desarrollar programas de aplicación. Un buen API hace más fácil desarrollar un programa proporcionando todos los bloques del desarrollo del mismo. El programador se encarga, entonces, de juntar los bloques. Entre los más importantes, de cara al dominio de los videojuegos, se destaca DirectX (de Microsoft) [Nicol Landa 2007] y OpenGL (que trabaja con la mayoría de los sistemas operativos) [Francis S. Hill 2007].

- Render (Renderización): Es la parte del código que se encarga del proceso de generar una imagen desde un modelo y ponerlo en la pantalla.

- Objetos 3D: Los objetos se almacenan por puntos en un mundo 3D, llamados vértices. Los vértices van formando polígonos. De esta manera, cuanto más polígonos posea un objeto, más complicado será éste, aunque también llevará más tiempo su renderizado. Es importante remarcar que el juego no necesita saber cuántos objetos hay en memoria o cómo el render va a mostrarlos: únicamente es necesario que el render los despliegue de la forma correcta, y que el modelo esté en el cuadro correcto de la animación.

- Culling: Proceso que logra que los objetos que no se vean en un determinado cuadro de la animación por causa de objetos que los obstaculizan (como una pared) no tomen tiempo de renderizado. Así se reduce la cantidad de trabajo del motor. Un método típico de Culling puede basarse en los "Árboles BSP" que se analizan a continuación [Hearn 2005].

- BSP Tree Hierarchy (Árbol de Jerarquía BSP): Método para determinar qué superficies de un mundo, y qué objetos, están realmente en la escena en momento dado, dada su localización en el mundo. Este método se utiliza a menudo para determinar los objetos de desecho, y también para omitir objetos, a fin de reducir el proceso del módulo de AI (Inteligencia Artificial) y del de la animación, tal y como se ha indicado anteriormente.

- Textura: componentes esenciales para que las escenas 3D sean reales. En sí las texturas son imágenes que se rompen en los distintos polígonos del modelo. Dado que muchas imágenes implicarán mucho gasto de memoria, es necesario usar técnicas de compresión:

    - Mapeo MIP: consiste en preprocesar las texturas creando múltiples copias de sus elementos cada una la mitad de la anterior. Con ello cada Texel (elemento de Textura) ocupa menos espacio.
    - Texturas Múltiples: requiere múltiples renderizados, por lo que para obtener buen resultado se necesita una tarjeta con Acelerador de Gráficos. Proporcionan mejor calidad que el simple mapeo. Se puede colocar una imagen sobre otra (más transparente) para dar el sentido de movimiento, pulso, o hasta sombra.
    - Antialiasing: El anti-aliasing revisa los polígonos y difumina las bordes y vértices, para que los bordes no se vean como dentados. Esta técnica se puede llevar a cabo de dos maneras. La primera se realiza de modo individual, entremezclando polígonos para sobreponerlos unos delante de otros. La segunda manera se hace por medio de tomar todo el marco y quitarle los bordes dentados, pero esto requiere de mucha memoria.

- Iluminación: Creación de luces de diversos tipos puntuales, direccionales en área o volumen, con distinto color o propiedades. Esto es la clave de una animación dentro de un videojuego. A continuación se presentan algunas técnicas para llevar a cabo dicha iluminación.

    - Vertex Lighting: Se determinan cuantos polígonos cruzan el vértice, se toma el total de todas las orientaciones de los polígonos (Normal) y se asigna la normal al vértice. Para cada vértice, un polígono dado reflejará la iluminación en una forma levemente distinta. La

ventaja es que al hardware le toma menos tiempo el procesarlo, pero este tipo de iluminación no produce sombras.

- Flat Shading Lighting (Iluminación de Sombreado Plano): Esta técnica calcula la apariencia final de un polígono a partir del ángulo existente entre la normal a la superficie del polígono y el rayo de luz incidente, la intensidad de ésta última, y los colores tanto del polígono como de la fuente de iluminación. Existen dos maneras de trabajar con el sombreado:
  - Vertex Shading (Sombreado de Vértice, Gouraud shading): solicita al motor de renderizado un color para cada vértice. Luego por medio de interpolación, se renderiza cada píxel por la distancia en relación con su respectivo vértice.
  - Phong Shading: técnica similar al Gouraud Shading. Ambas trabajan con la textura, con la salvedad de que el Phong Shading usa los píxeles en lugar de los vértices. El Phong Shading toma más tiempo de procesamiento que el Vertex Shading, pero sus resultados son mucho mejores en cuestión de suavizado de texturas.

- Light Map Generation (Generación del mapa de luz): Esta técnica usa una segunda capa de textura (mapa de luz) que dará el efecto de iluminación a los modelos. Con ello se consigue un efecto excelente, pero debe llevarse a cabo antes del renderizado. No obstante, si se tienen Luces Dinámicas (o sea luces que se mueven, encienden o apagan sin intervención de programa) se debe regenerar los mapas en cada frame de animación, lo que implica mucho gasto de memoria. La ventaja es que acelera el renderizado.

- Scripting Systems (Sistemas de scripting): Los sistemas de scripting son lenguajes de programación dinámicos, embebidos en el motor del juego, que permiten al diseñador configurar los aspectos de alto nivel del mismo (p.ej., controlar y manipular la escena, colocar distintos objetos, manejar distintos eventos, etc.). La ventaja de su uso radica en que facilitan la intervención, en el proceso de programación, de un diseñador que no tiene que ser necesariamente un experto en los detalles de bajo nivel del motor (p.ej., iluminación, renderizado, etc.), sino que únicamente debe preocuparse de los aspectos más lógicos del juego.

- Físicas: Consiste en un tipo de programación, en donde se supone la introducción de las leyes de Física en un simulador o motor de juego. Normalmente afecta a los gráficos 3D por computadora, aunque también existe en gráficos 2D. El propósito es hacer que los efectos físicos de los objetos creados o modelados tengan las mismas características que en la vida real, teniendo en cuenta, por ejemplo, gravedad, masa, fricción, restitución, etc. Los medios más utilizados son:

    - El Motor físico: que es un código de programa usado para simular la Mecánica newtoniana en el ambiente.
    - El Detector de colisiones: que se utiliza para resolver el problema de determinar cuándo dos o más objetos físicos en el ambiente se cruzan entre sí.
    - Física Ragdoll: Sistema que interpreta las características del cuerpo como una serie de huesos rígidos conectados entre sí, poniéndole bisagras para simular las articulaciones. La simulación modela lo que ocurre con el cuerpo cuando se cae al suelo, como si fuese un muñeco de trapo (de ahí el nombre Ragdoll). Un modelo físico del movimiento del cuerpo y una interacción con la colisión más sofisticada requiere una potencia de cálculo mucho mayor, ya que se necesita hacer una simulación más exacta del los sólidos, líquidos e hidrodinámica. El sistema del modelo articulado puede también reproducir los efectos del esqueleto, músculos, tendones y otros componentes fisiológicos.

- Sistema de Partículas: Las partículas son un aspecto común en los videojuegos en donde se trata de recrear algún tipo de explosión en cada circunstancia. Estos sistemas permiten emular una amplia variedad de otros fenómenos físicos, incluyendo humo, agua en movimiento, precipitaciones, etc.

- Sonido: Componente que se encarga de manejar todos los efectos sonoros y musicales del juego. Cabe destacar en este campo del sonido a OpenAL [Charles River 2006], que es un API para los sistemas de los sonidos de la misma manera que OpenGL es un API para los gráficos.

- Inteligencia Artificial: componente que se encarga de gobernar el comportamiento de los distintos personajes del juego, de manera que éste

parezca intencionado e inteligente. Este componente es crítico para garantizar una forma de juego (game play) ágil y atractiva.

2.3.2. Ejemplos de motores

2.3.2.1. Unity

El motor de juego Unity[2] tiene como principal ventaja una total integración con el entorno de desarrollo. Esta estrecha relación permite que el editor sea fácil de manejar y el desarrollo del juego sea mucho más cómodo. El editor permite reproducir el juego e ir avanzando y parando en las distintas escenas para poder observar y cambiar los valores que sean necesarios. Cabe destacar algunas características del editor como pueden ser: asignación de texturas, audio, creación de conductas y secuencia de comandos y la integración de lógica para gestionar y maximizar la funcionalidad del juego. Para evitar el uso repetido de GameObjects (elementos de los que se compone la escena), se puede activar uno o más de ellos en un Prefab (componente similar a una Interface). Este Prefab se puede colocar fácilmente en todo el juego y se puede instanciar en tiempo de ejecución. Cualquier cambio en el Prefab original se propaga a todos los dependientes. Como consecuencia, cualquier tipo de ajuste se puede propagar de manera rápida a un gran número de objetos del juego.

Principales características del motor

Gráficos

En la sección de gráficos se destaca el rendering, que está diseñado para reducir al mínimo los cambios de estado, teniendo en cuenta las luces y sombras. Unity es capaz de procesar millones de polígonos por segundo. De esta manera, asegura que sus juegos funcionan en todos los entornos. Para ello incluye DirectX y un renderizador de OpenGL. La creación visual y la manipulación de los sistemas de partículas es simple, permitiendo la creación de: lluvia, chispas, estelas de polvo, etc.

[2]http://unity3d.com

Importación de archivos

Unity puede importar modelos 3D, huesos, y las animaciones de casi todas las aplicaciones 3D. Soporta Maya [Derakhsani Dariush 2008], 3DS Max [3DSMAX 2010], Cinema 4D [Koenigsmarck Arndt Von 2007], Cheetah3D[3] y Blender [Mullen Tony 2007]. Respecto al audio Unity soporta cualquier formato de audio que sea compatible con QuickTime [Bradley Ford 2003]. El audio puede ser convertido internamente.

Soporte

El motor Unity proporciona soporte para Mac OS X y Windows 2000/XP/ Vista/7. Los juegos implementados con Unity se pueden ejecutar en un navegador web a través de un plug-in. Por último Unity también permite realizar juegos para IPhone y Wii.

Sombras

Referente al sistema de sombras de Unity, éste combina la facilidad de uso con la flexibilidad y rendimiento. Todas las sombras se integran perfectamente con cualquier tipo de luz. Por último también ofrece la posibilidad de implementar un sistema de sombras.

Terreno

Unity ofrece herramientas en el editor para poder generar terrenos montañosos. Posee también texturas en mosaico que pueden ser mezcladas y combinadas con una colección de herramientas de precisión. Esto permite tener una gran variedad de texturas de baja resolución para poder crear terrenos variados. Por último, se da la posibilidad de generar un mapa de luces para el terreno en cualquier momento. Con ello se calcula el efecto de todas las luces direccionales en el paisaje de una forma eficaz , lo que permite una rápida configuración. Unity combina la extrema facilidad de uso con un alto rendimiento.

[3]http://www.cheetah3d.com/index.php

Red

Unity permite llevar de forma cómoda el juego a un modo multijugador. Se puede crear un sistema de puntuación en Internet, introducir un chat, dar de alta un nuevo usuario en una base de datos, etc.

Física

Unity contiene todas las capacidades del motor de física AGEIA PhysX Next-Gen[4]. También se apoya en la física Rigid Body [Jenkins López David 2006] para las fuerzas, choques, y el modelado del trabajo de las uniones de piezas sin requerir, para ello, scripting.

Sonido

Mezcla en tiempo real gráficos en 3D con streaming de audio y vídeo. Reconoce multitud de formatos de audio y vídeo.

Lenguaje

Unity es compatible con tres lenguajes de scripting: JavaScript [GoodMan Danny 2008], C # [Liberty Jesse 2008], y un dialecto de Python llamado Boo[5]. Los tres pueden utilizar bibliotecas .NET que ofrecen integración con las bases de datos, expresiones regulares, XML, acceso a archivos y redes.

[4]http://www.nvidia.com
[5]http://boo.codehaus.org

Figura 2.3: Algunas capturas de juegos creados con Unity (tomadas de la página oficial de Unity).

2.3.2.2. Ogre

Características generales

Ogre[6] es un motor gráfico simple, fácil de usar, y que ofrece una interfaz diseñada para minimizar el esfuerzo requerido para realizar las escenas 3D, independiente de la aplicación 3D que se quiera realizar.

Dicho motor se ayuda de frameworks extensible para conseguir una aplicación en ejecución mas rápida y sencilla.

Los aspectos comunes de los motores gráficos, como llevar a cabo una buena gestión de render, la eliminación selectiva del terreno y la transparencia, se realizan de forma automática. Todo ello conlleva un ahorro de tiempo.

Limpieza, diseño robusto y documentación completa de todas las clases del motor son señas de identidad de Ogre.

Plataforma de apoyo y 3D API

Ogre incluye soporte Direct3D y OpenGL.

Plataformas

Ogre ofrece servicio en Windows (todas las versiones principales), Linux y Mac OSX.

Lenguaje

Visual C# [Sharp John 2010] y Code::Blocks[7] en Windows. Gcc 3+ [Wall Kurt] en Linux / Mac OSX (utilizando XCode [James Bucanek 2006]).

Materiales / Sombras

Ogre cuenta con un poderoso lenguaje de declaración de materiales que permite mantener los materiales fuera del código. Por otro lado, es compatible con las operaciones de función fija, como multitextura y multipaso de mezcla, coordinando la generación de texturas y su modificación, independientemente del color y de las operaciones alfa. Ogre incluye soporte para múltiples técnicas de materiales. Con ello ofrece varias alternativas para una

[6] http://www.ogre3d.org

[7] http://www.codeblocks.org

amplia gama de tarjetas. Permite la carga de texturas en formato PNG, JPEG, TGA, BMP o archivos DDS. Las texturas pueden ser proporcionadas y actualizadas en tiempo real por los plugins (por ejemplo un canal de video).

Animación

Ogre incluye una sofisticada ayuda relativa a la animación del esqueleto. Para ello se ayuda de una animación flexible en la que se posibilita la elección de la postura de la animación, permitiendo combinar muchas poses con pesos variables a lo largo de una línea de tiempo. Ogre incluye animación de SceneNodes para las rutas de la cámara. El sistema de animación cuenta con pistas de animación genéricas para poder utilizar objetos en los que se pueda animar cualquier parámetro de cualquier objeto durante un periodo de tiempo.

Escena

Ogre permite una escena altamente personalizable, utiliza clases predefinidas para la organización de la escena y ofrece control total sobre la organización de las escenas. La escena cuenta con una representación en forma de grafo jerárquico, los nodos del cuál permiten que los objetos se adjunten a los demás, posibilitando, de esta forma, seguir los movimientos de los mismos.

Efectos especiales

Por último Ogre contiene un compositor que permite el completo postprocesado de los efectos a través de secuencias de comandos y un sistema de partículas para ser utilizado en los efectos especiales.

Figura 2.4: Algunas capturas de juegos creados con Ogre (tomadas de la página oficial de Ogre).

2.3.2.3. Irrlicht

Irrlicht[8] es una plataforma 3D de alto rendimiento escrita en C ++ [Enrique Hernández Orallo 2001]. Cuenta con un alto nivel para crear en 3D y 2D aplicaciones completas, como juegos o visualizaciones científicas. Viene acompañada de una excelente documentación e integra todos los estados de las funciones más novedosas para la representación visual, como sombras dinámicas, sistemas de partículas, animación de personajes, tecnología de interior y exterior, y la detección de colisiones. Todo esto es accesible a través de una interfaz bien diseñada en C ++, que es muy fácil de usar.

Efectos especiales

Existe un gran variedad de efectos especiales comunes disponibles en el motor. El motor ofrece soporte para los efectos especiales más comunes (animación de agua, luz dinámica, sombras dinámicas, objetos transparentes, mapas de iluminación, sistemas de partículas, animación de textuales, niebla, etc.). En la mayoría de los casos el programador sólo tiene que invocarlos. El motor se amplía continuamente con nuevos efectos.

Drivers

El motor Irrlicht soporta seis APIs de representación: Direct3D 8.1, Direct3D 9.0, OpenGL 1.2-3.x, motor de renderizador de software de Irrlicht, renderizador de software Burningsvideo y un dispositivo nulo. Al utilizar el motor Irrlicht, el programador no necesita conocer la API del motor que está utilizando; sólo tiene que indicar a la máquina que API del motor prefiere.

Materiales y Sombras

Con el motor Irrlicht se pueden crear entornos realistas, existiendo para ello una gran variedad de materiales construidos disponibles en el motor. Algunos de los materiales se basan en la tubería de función fija y algunas se basan en la tubería programable que hoy el hardware 3D ofrece. Sin embargo, si los materiales construidos no son suficientes, es posible añadir nuevos materiales para Irrlicht en tiempo de ejecución, sin necesidad de modificar o volver a compilar el motor.

[8] http://irrlicht.sourceforge.net

Plataformas

El motor de Irrlicht es independiente de la plataforma. Actualmente el motor funciona en: Windows 98, ME, NT 4, 2000, XP, XP64, Vista, Linux, OSX, Sun Solaris / SPARC y todas las plataformas usando SDL[9].

Escena

El rendering en el motor Irrlicht se realiza utilizando un escenario gráfico jerárquico. Los nodos de la escena están unidos entre sí, siguen los movimientos de los demás y son capaces de realizar la detección de colisiones. El motor puede mezclar perfectamente escenas en interiores y exteriores en conjunto, proporcionando al programador control total sobre lo que está sucediendo en la escena. Es fácilmente extensible, porque el programador es capaz de añadir su propia escena, nodos, mallas, cargadores textura y elementos de la GUI. El creador de geometría proporciona un fácil acceso a los cuerpos simples geométricos, tales como cilindros, cubos, etc, objetos que pueden ser renderizados como polígonos o puntos, usando triángulos, líneas, puntos y primitivas.

Animación del personaje

Actualmente hay dos tipos de animación de los personajes:

- Animación Morph: Las mallas se interpolan linealmente a partir de un cuadro a otro. El motor Irrlicht lo hace cuando se importan archivos .MD2 y .MD3.
- Animación del esqueleto: La piel se manipula por las articulaciones de animación. El motor Irrlicht lo hará cuando se cargan Ms3d y los archivos B3d. Es posible adjuntar objetos a partes del modelo de animación. El programador no necesita saber acerca de animaciones; sólo necesita realizar el diseño, cargar los archivos en el motor y dejar que se animen.

Formatos Soportados

Muchos de los formatos de los archivos comunes son compatibles, y son capaces de ser cargados directamente desde el motor. De esta manera, los

[9]http://www.libsdl.org

datos nunca necesitan ser convertidos para su utilización, por lo que se ahorra tiempo de desarrollo. La gestión interna de recursos de Irrlicht proporciona un acceso sencillo a todos los formatos de archivo y se encarga de buscar las mallas ya cargadas o texturas de su propia caché de disco.

Figura 2.5: Algunas capturas de juegos creados con Irrlicht (tomadas de la página oficial de Irrlicht).

2.3.2.4. Horde3D

Horde3D[10] es un potente motor de gráficos diseñado para satisfacer las necesidades de los juegos compatible con cualquier plataforma mediante el uso de OpenGL. Dicho motor está orientado a C ++. El código para el desarrollo contiene una gran modularidad y alta abstracción. Por otro lado, ofrece fácil integración con los motores de juego y motores de física, debido al diseño no intrusivo de la API. Horde3D se adapta bastante bien para aplicaciones de Realidad Aumentada [Stephen Cawook 2008].

Escena y Gestión de Recursos

Horde3D posee una interfaz para la carga de datos de cualquier tipo de archivo. Para optimizar dicho proceso, el motor recarga los recursos para aumentar la productividad durante el desarrollo. Horde3D posee una estructura jerárquica de la escena. La escena se representa como un sistema unificado, los modelos y los esqueletos son ramas del grafo de la escena.Estas ramas se puede cargar y guardar a través de XML con las funciones de la API. Por otro lado se da la posibilidad de conectar los nodos de escena a las articulaciones de los personajes. Además, se tiene acceso a los datos de los vértices para la detección de colisiones y la interoperabilidad con los motores de la física.

Rendering

El motor Horde3D implementa generación automática mediante permutación de sombreado. Cuenta también con un canal de renderizado personalizable para las pruebas de las diferentes técnicas de representación y postprocesamiento de los efectos, como el desenfoque de movimiento. En el motor coexisten diferentes técnicas de sombreado diferido. Horde3D mantiene soporte para casi todas las técnicas modernas de representación. Cuenta también con la posibilidad de ofrecer reflexiones en tiempo real y otras técnicas que requieren de varias cámaras para su desarrollo. Por último incluye un sistema de partículas completamente integrado que puede arrojar sombras.

[10]http://www.horde3d.org

Animación

El sistema de animación de Horde3D trabaja directamente en la escena gráfica. Contiene animaciones de fotogramas clave para las articulaciones y mallas. Cabe destacar la existencia de animaciones del esqueleto de cuatro pesos por cada vértice de los modelos articulados. También se ofrece animaciones por mezcla de capas y el uso de máscaras y canales aditivos. Así mismo, el motor utiliza la animación Morph para la animación facial y sincronización de los labios de los personajes. Por último, para las animaciones de fluidos se apoya en las interpolaciones.

Content Pipeline

Horde3D trata el modelo y la animación mediante formatos optimizados para un máximo rendimiento. Para ello mezcla formatos binarios y XML para un mejor equilibrio entre el rendimiento y la productividad. Soporta entre otros: texturas DDS y otros formatos de imágenes comunes. Utiliza Collada Converter [Rémi Arnaud 2006] para convertir los archivos a un tipo compatible con Horde3D. Este convertidor se implementa como herramienta de línea de comandos que puede ser integrada en el proceso de generación automática. Por último, se destaca el potente editor que posee para componer escenas y desarrollo de sombras.

Figura 2.6: Algunas capturas de juegos creados con Horde3D (tomadas de la página oficial de Horde3D).

2.3.2.5. Unreal Engine 3

Unreal Engine 3 se engloba en un marco de desarrollo para DirectX para equipos como PC, Xbox 360 y PLAYSTATION 3. Ofrece una amplia gama de tecnologías básicas, herramientas de creación de contenidos e infraestruc-

tura de apoyo que requieren los juegos. También permite la visualización avanzada de la simulación y la creación de contenido lineal de animación 3D. Cada aspecto del Unreal Engine ha sido diseñado con el objetivo de facilitar el desarrollo de juegos con un entorno visual. El entorno ofrece a los programadores una gestión altamente modular, escalable y extensible para la construcción y prueba de juegos.

El renderizador de Unreal Engine

Unreal Engine soporta un rango de 64-bit de color en el canal de renderizado. Incluye también soporte para las iluminaciones por píxel y técnicas de renderizado. Por otro lado cabe destacar la gran iluminación detallada del personaje. En dicha iluminación se incluyen también sombras avanzadas, proporcionando un soporte completo para técnicas de sombreado. Todas las técnicas de sombras soportadas son visualmente compatibles y pueden mezclarse libremente a discreción del artista. También pueden combinarse con las funciones de atenuación de colores para generar un foco direccional, sombras y efectos de luz del proyector. Este motor incluye efectos volumétricos del medio ambiente que se integran perfectamente con los entornos. Otro aspecto a tener en cuenta es el soporte que brinda para ambientes interiores y al aire libre sin problemas de interconexión con la iluminación dinámica por píxel y el sombreado apoyado por todas partes. En lo que se refiere a las herramientas de textura, se permiten reflexiones en tiempo real dinámico, así como la captura de escenas estáticas. Se ofrece también un constructor de terreno y una herramienta de edición. Por último destacar que tanto el sombreado como el sistema de partículas incluyen un sistema de gestión y desarrollo para facilitar el diseño del juego.

Sistema de animación Unreal

Unreal incluye un componente denominado AnimSet Visor, herramienta para la navegación y la organización de animaciones y mallas. Dichas animaciones están impulsadas por un AnimTree, árbol de nodos de la animación. AnimTree Editor permite a los programadores crear mezclas complejas, configuraciones del controlador y vistas previas en tiempo real. Con ello se proporciona la facilidad de agregar nuevas animaciones a los nodos y a los controladores del hueso. Referente al tema de animaciones, Unreal soporta Animación Morph. Este tipo de animación permite el control de pe-

sos de la mezcla en los datos de animación, con soporte para control de los materiales. Puede ser escrito en malla, y los datos pueden ser exportados. Se puede controlar la animación de vértices y materiales, con vista preliminar en el Visor de AnimSet. Uno de los puntos fuertes de Unreal es el seguimiento del uso de la animación. De esta manera se puede obtener estadísticas sobre las animaciones que son visibles durante el juego. Todo ello ayuda a ver las pistas de las animaciones que no se han utilizado, y también para poder ver cuáles son las más visibles para el jugador. Unreal cuenta con herramientas de exportación para 3D Studio Max, Maya y XSI para importar mallas ponderada, esqueletos y secuencias de animación. Una vez cargada la animación, se le puede dar movimiento a la unidad física del personaje en el juego. En Unreal una animación puede conducir la física de un personaje, y permite la colisión con otros objetos.

Sistema de audio de Unreal

Unreal ofrece soporte para los últimos sistemas de compresión de audio en todas las plataformas. Cuenta con modos de sonido para controlar el paisaje sonoro del juego, controles de tono, volumen, atenuación, actores de sonido, grupos de sonido, la compresión y otros parámetros. Por último ofrece herramientas de depuración que permiten supervisar el uso extensivo de los recursos y la optimización de recursos.

Sistema de física de Unreal

La física de Unreal se basa en GPU NVIDIA PhysX[11], siendo éste un sistema rígido de física con soporte para la interacción del jugador con los objetos físicos del juego. Además contiene animación de personajes (incluyendo el control del jugador, IA, y la creación de redes) y vehículos complejos. Unreal incluye un módulo de física de materiales, un sistema que permite incorporar propiedades a la superficie, como la fricción, sonidos y efectos. Por otro lado se destaca la inclusión del componente UnrealPHAT, herramienta visual de modelado de física que soporta la creación de primitivas de colisión optimizando modelos de mallas y el esqueleto animado, así como la edición de restricción y simulación interactiva de física, pudiéndose ajustar todo ello en el editor. Todos estos aspectos permiten contar con entornos destructibles en la escena, que se resquebrajan de manera realista. Se destaca también la

[11]http://www.nvidia.com

capacidad de alta densidad de multitud con la que cuenta Unreal: se puede simular en tiempo real cientos de personajes dentro de una escena.

Efectos visuales de partículas en el sistema

Unreal posee una sencilla herramienta utilizada por los artistas y diseñadores para crear efectos visuales para los juegos y secuencias de cine. Dicha herramienta permite incluir efectos visuales en tiempo real, permitiendo realizar, por ejemplo, ajustes dinámicos a un efecto como una explosión o un fuego ardiente, sin tiempo de demora en la respuesta. Por último, cabe destacar la capacidad de interactuar con diferentes sistemas para crear una interfaz intuitiva y eficiente para la libertad de creación.

El editor de Unreal

Unreal cuenta con una potente herramienta de creación de contenido para llenar el vacío de las herramientas de creación de formatos como 3DS Max y Maya. El editor es, en realidad, un conjunto de herramientas diversas para la realización de contenido en el Unreal Engine. Las características incluyen:

- Arte visual para la colocación y edición de objetos de juego, tal como los jugadores, los artículos del inventario, los nodos de ruta IA y fuentes de luz.
- Renderizado totalmente interactivo en tiempo real durante la edición: todo el contenido en el editor se presenta de manera coherente con el motor del juego.
- Permite a los diseñadores personalizar fácilmente cualquier objeto del juego, y proporciona la posibilidad de desarrollar nuevas propiedades personalizables a través de secuencias de comandos.
- Por último destacar las características de la herramienta de edición de contenidos de Unreal. En particular, cabe destacar: Multi-nivel deshacer / rehacer, arrastrar y soltar, agrupación de objetos, guardado automático , controles de cuadrícula, zoom en el cursor en los visores, el movimiento de cámara estándar o invertida, de copiar y pegar, personalizar las configuraciones de visualización, el teclado y personalización de colores, etc.

Figura 2.7: Algunas capturas de juegos creados con Unreal Engine 3 (tomadas de la página oficial de Unreal).

2.3.2.6. JMonkey

JMonkey[12] es el motor utilizado en el desarrollo de Knuthians. Este motor proporciona una API de alto rendimiento para objetos 3D; es decir, es un conjunto de librerías para programar juegos 3D en Java. Usa una capa de abstracción para el renderizado 3D, lo que en teoría permite el uso de cualquier motor de render. Cabe destacar que soporta una abstracción denominada LWJGL.

LWJGL

La Lightweight Java Game Library (LWJGL o Biblioteca Java Ligera para Juegos) es una solución dirigida a programadores, tanto amateurs como profesionales, y está destinada a la creación de juegos de calidad comercial escritos en el lenguaje Java. LWJGL proporciona a los desarrolladores acceso a diversas bibliotecas multiplataforma, como OpenGL (Open Graphics Library) y OpenAL (Open Audio Library), permitiendo la creación de juegos de alta calidad con gráficos y sonido 3D.

Por otro lado, LWJGL permite además acceder a controladores de juegos como gamepads, volantes y joysticks. El auténtico objetivo de LWJGL no es crear un motor gráfico que permita crear juegos espectaculares de forma casi inmediata, sino que lo que pretende es dar acceso a los programadores Java a una tecnología y unos recursos que normalmente no se implementan correctamente en dicho lenguaje. Por tanto, LWJGL debe entenderse más bien como una API base sobre la que en la actualidad ya se están apoyando algunas potentes herramientas gráficas, como es el caso de la API de scenegraph jMonkey.

Caracteristicas

Tanto jME como LWJGL están disponibles bajo licencia BSD (Berkeley Software Distribution) y por lo tanto son de libre distribución. A esta característica hay que unirle su potencia y su relativa facilidad de uso, lo que convierte a jME en una muy buena herramienta a tener en consideración. Lamentablemente como software libre en fase de desarrollo presenta continuamente revisiones que añaden y corrigen fallos. Aunque su filosofía no es muy

[12]http://www.jmonkeyengine.com

compleja, es necesario conocimientos básicos de entornos 3D, técnicas de representación y estar familiarizado con OpenGL para poder sacar el máximo partido a las posibilidades que nos ofrece. Y como cualquier motor gráfico requiere un mínimo de tiempo para llegar a entenderlo y poder usarlo.

La arquitectura en la que está basada jME permite la organización de los datos de la aplicación en forma de árbol, donde el nodo padre puede tener cualquier número de nodos hijos (hojas), pero un hijo solo puede tener un padre. Esta organización está pensada para una fácil gestión de los elementos en una escena y poder realizar procesamientos rápidos de tareas (por ejemplo mostrar sólo todo lo que depende del nodo padre X y ocultar el resto de la escena). Los nodos hojas se denominan geometrías (Geometry), las cuales pueden ser renderizadas (mostradas) en pantalla. jME dispone de varias geometrías: curvas Bezier para controlar el nodo, líneas (Line), puntos (Points), modelos (MD2, ASE, etc.), terrenos (Terrain) y algunas más. Además jME soporta efectos de alto nivel como: Sistemas de partículas (Particle Systems) y destellos en la lente (Lens Flare).

A continuación se detallan algunos aspectos técnicos del motor:

- Total integración con Java Applet, AWT y Swing, SWT (Standard Widget Toolkit) y la utilización del lenguaje de marcado XML.

- Existencia de un bucle principal de juego, SimpleGame. Este proporciona un punto de acceso rápido para la prueba de conceptos: StandardGame y GameStates que ofrecen la posibilidad de cambiar los estados del juego (menú, juego, créditos, etc).

- Referente al sistema de iluminación, el motor admite hasta ocho luces a la vez y da la opción de la selección óptima de la luz. Soporta luz direccional y luz de punto. También se incluye un sistema de sombras dinámicas.

- El sistema de cámaras mantiene una cámara como un objeto independiente o un nodo en la escena. Utiliza la técnica de Frustum [Francis S. Hill 2007] para ser más eficiente.

- En la parte de Renderer, JMonkey ordena los elementos de la escena sobre la base de la opacidad. Los opacos son ordenados de adelante hacia atrás y puestos en primer lugar. A continuación se colocan los transparentes ordenados de atrás para adelante. En el render la textura

se utiliza como un modelo único y puede ser animada tan lentamente como sea necesario para aumentar la velocidad de fotogramas.

- Por último destacar, en sonido, la existencia de soporte para OpenAL y la ayuda que JMonkey ofrece con el álgebra lineal con la librería matemática (incluyendo, por ejemplo, completo soporte para el manejo de rotaciones y translaciones mediante cuaternios).

Figura 2.8: Algunas capturas de juegos creados con JMonkey (tomadas de la página oficial de JMonkey).

2.3.3. Conclusiones

El cuadro 1 muestra una tabla comparativa entre los diferentes motores estudiados anteriormente.

En una primera valoración se descartan los motores que no incorporan físicas, ya que el juego contiene una gran componente física para el movimiento del personaje. Pasada la primera restricción quedan como posibles motores Unity, Unreal y JMonkey. De estos tres motores, Unity no es multiplataforma por lo que también queda descartado. Unreal y JMonkey son los dos grandes candidatos para ser usados. Cabe destacar que Unreal incorpora un completo editor para facilitar la elaboración del juego, mientras que JMonkey carece de ello.

Finalmente en el proyecto se ha utilizado el motor de JMonkey, ya que éste está completamente basado en Java. Debido a la componente educativa del proyecto, se espera que el uso de Java facilite el futuro alojamiento del proyecto en el campus virtual de la universidad.

	Unity	Ogre	Irrlicht	Horde3D	Unreal Engine 3	JMonkey
Plataforma	Wi/Ma	Wi/Li/Ma	Wi/Li/Ma	Wi/Li/Ma	Wi/Li/Ma	Wi/Li/Ma
Gráficos	DX/OGL	DX/OGL	DX/OGL	OGL	DX	LWJGL
Sonido	Si	No	No	No	Si	Si
Lenguaje	C#/JS	C	C++	C++	C++	Java
Física	Si	No	No	No	Si	Si
Editor	Si	Si	Si	Si	Si	No
Efectos	Si	Si	Si	Si	Si	Si

Cuadro 2.1: Tabla comparativa.

En la tabla las abreviaturas utilizadas son las siguientes Wi: Windows, Li: Linux, Ma: Mac OS X, DX: Direct3D, OGL: OpenGL y JS: JavaScript.

2.4. Lenguajes de Marcado

Los lenguajes de marcado consisten en una manera de codificar un documento de tal forma que éste contenga etiquetas o marcas que proporcionan información acerca de la estructura del texto o de su presentación. Algunos ejemplos de este tipo de lenguajes son: SGML, HTML o XML.

A continuación se pasa a describir de forma concreta XML, por ser el utilizado en el proyecto como soporte de almacenaje para los ejercicios que se presentan en Knuthians.

2.4.1. XML

XML (e**X**tensible **M**arkup **L**anguage)[David Hunter 2007] es un metalenguaje extensible de marcado. Ha sido desarrollado por el World Wide Web Consortium (W3C).

La principal ventaja de XML es su gran versatilidad. XML es una herramienta utilizada en todo tipo de aplicaciones y para multitud de finalidades. Esto incluye desde la comunicación de dos ordenadores a través de una red o Internet, hasta la persistencia de datos, pasando por la definición de la estructura de un tipo de documento.

XML surgió como una simplificación y adaptación del SGML (**S**tandard **G**eneralized **M**arkup **L**anguage)[Charles F.Goldfarb 1998]. El lenguaje HTML está definido en términos del SGML, ya que la normalización de XML fue posterior al diseño del lenguaje de marcas HTML.

Además, XML cuenta con el concepto de "documento bien formado" y "validez" como elementos separados:

- Documento bien formado: Es aquel documento que cumple todas las definiciones básicas del formato, y, puede por tanto, ser analizado correctamente por cualquier analizador sintáctico que cumpla con los estándares.

- Documento válido: Es aquel documento que además de estar bien formado sintácticamente, semánticamente es correcto, o aquello que describe tiene sentido. Para que la comprobación de validez se pueda realizar, se necesitará un modelo gramatical del documento, que exprese las relaciones entre los diferentes elementos del documento XML, y que establezca unos límites para los valores de los atributos. Como un ejemplo, un documento podría considerarse inválido si guardara en un campo fecha un valor de "31 de febrero de 2010". La gramática documental sería la encargada de fijar los límites de este campo "fecha". XML incluye un formalismo de gramática documental denominado DTD (Document Type Definition). No obstante, se han propuesto otros formalismos con mayor poder expresivo, como XML Schema [Eric van der Vlist 2002].

Las partes de un documento XML están bien definidas, y son:

- Prólogo: Contiene información acerca del documento XML. Describe la versión de XML, el tipo de documento, la codificación utilizada, y otro

tipo de metainformación. Es opcional para que un documento esté bien formado.

- Cuerpo: El cuerpo debe contener un único elemento raíz para que el documento esté bien formado. Es obligatorio para que un documento esté bien formado.
- Elementos: Los elementos pueden contener: elementos, caracteres o ambos.
- Atributos: Los elementos pueden contener atributos, que son una manera de añadir características o propiedades a los elementos de un documento. Deben ir entre comillas para que un documento esté bien formado.
- Entidades predefinidas: Entidades para representar caracteres especiales para que el analizador sintáctico de XML no los interprete como marcado. Por ejemplo, &amp es el símbolo &.
- Secciones CDATA: Es una construcción XML para especificar datos utilizando cualquier caracter sin que se interprete como marcado XML. Así se consigue que caracteres especiales no rompan la estructura XML.
- Comentarios: Comentarios a título informativo por parte del diseñador del XML, que el procesador de XML ignorará.

Además de todo lo expuesto, XML cuenta con una gran facilidad de transformación. XSLT, por ejemplo, es otro estándar desarrollado por el World Wide Web Consortium (W3C) que permite muy fácilmente transformar documentos XML y aplicarles estilos [Doug Tidwell 2008]. De esta manera, se separa el contenido de la presentación de una manera muy natural.

En general, XML cuenta con un ecosistema de herramientas y lenguajes anexos muy potentes que hacen que trabajar con este lenguaje de marcado sea lo idóneo en un gran número de ocasiones.

2.5. Gramáticas de Atributos

El juego pretende enseñar el funcionamiento del formalismo de las gramáticas de atributos. Este formalismo, propuesto por D.Knuth a finales de los sesenta, presenta un mecanismo para añadir semántica a las gramáticas incontextuales [Aho 2007 , Paaki 1995, D.E. Knuth 1968 y 1971]. La Figura 2.9 muestra un ejemplo de gramática de atributos. A continuación se explicará brevemente este formalismo explicando cada una de las partes que lo componen.

```
Expr ::= Expr + Term

        Expr0.val = Expr1.val + Term.val
        Expr1.tsh = Expr0.tsh
        Term.tsh = Expr0.tsh

Term ::= num

        Term.val = str2int(num.lex)

Term ::= Expr

        Term.val = Expr.val

Term ::= id

        Term.val = findVar(id.lex,Term.tsh)
```

Figura 2.9: Ejemplo de gramáticas de atributos.

Una gramática de atributos consta de:

- Una gramática incontextual, que consta a su vez de los siguientes elementos:
  - Símbolos terminales: son los símbolos básicos a partir de los cuales se forman las cadenas o sentencias.
  - Símbolos no terminales: son variables sintácticas que denotan conjuntos de cadenas. Los conjuntos de cadenas denotados por los no

terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre las cadenas del lenguaje, hecho que aprovecha nuestro juego para presentar dichas estructuras como un laberinto donde el jugador debe moverse.

- Producciones : especifican la forma en que pueden combinarse los terminales y los no terminales. Cada producción tiene la estructura que aparece en la figura 2.10.

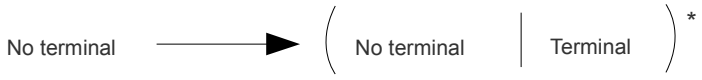

Figura 2.10: Estructura de una producción.

- Un conjunto de atributos que se asocian a los símbolos de la gramática. Existen dos tipos de atributos:

  - Atributos sintetizados : son atributos cuyos valores se calculan a partir de los valores de los atributos que se encuentran en los no terminales a los que están asociados y de los hijos de estos (véase el diagrama de flujo de la figura 2.11, donde se ilustra el flujo de los atributos sintetizados de la gramática de la figura 2.9 para una sentencia de ejemplo).

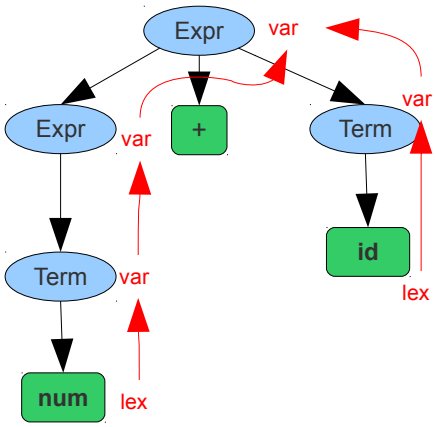

Figura 2.11: Diagrama de flujo de los atributos sintetizados.

- Atributos heredados : son atributos cuyos valores se calculan a partir de los valores de los atributos que se encuentran en los no terminales a los que están asociados, de los que se encuentran en los padres de los no terminales a los que están asociados y de los atributos de los hermanos de los no terminales a los que están asociados (véase el diagrama de flujo de la figura 2.12, donde se ilustra el flujo de los atributos heredados de la gramática de la figura 2.9 para el ejemplo anterior).

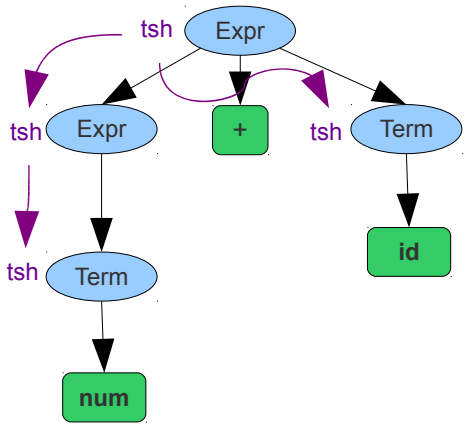

Figura 2.12: Diagrama de flujo de los atributos heredados.

- Un conjunto de ecuaciones semánticas asociadas a cada producción de la gramática, donde se especifica cómo se calcularan los valores de los atributos sintetizados asociados con la parte izquierda, y de los atributos heredados asociados con cada no terminal de la parte derecha. Para ello, se aplican funciones semánticas sobre los atributos implicados en dicho computo.

Durante la especificación de una gramática de atributos no es necesario especificar explícitamente el orden en el que se aplicarán las ecuaciones semánticas para calcular los valores de los atributos. El orden está implícito en las dependencias entre los atributos introducidas en las ecuaciones semánticas. Nuestro juego, de hecho, se basa en retar al alumno a que encuentre órdenes de propagación válidos para una gramática y una frase de ejemplo.

Capítulo 3

Knuthians

3.1. Introducción

Knuthians es un videojuego que pretende enseñar el funcionamiento interno del paradigma de las gramáticas de atributos presentado por D.Knuth. Para llevar a cabo este cometido el juego propone resolver un laberinto creado a partir de un árbol sintáctico de una sentencia. El usuario deberá moverse por este laberinto resolviendo las dependencias de los atributos del árbol sintáctico. Al ser un proyecto orientado a la educación, el proyecto está dividido en dos partes muy diferenciadas e íntimamente ligadas:

- Por una parte se tiene el juego en sí, más orientado al alumno, que constará de las herramientas necesarias para poder cargar distintos mapas y poder resolverlos.
- Y por otro lado habrá unas herramientas de edición y creación de escenarios orientadas a los tutores.

A continuación se describen en mayor detalle las características de ambas partes.

3.2. El videojuego

Knuthians es un videojuego en tercera persona donde el usuario se pone en la piel de un avatar para recorrer un laberinto y mover objetos entre las

distintas estancias de este laberinto con el objetivo final de activar todos estos objetos.

A lo largo de esta sección se explicarán cada una de las distintas pantallas e interfaces que el usuario encontrará mientras use Knuthians. Seguidamente, se describirán en profundidad cada uno de los elementos con los que el usuario podrá interactuar a lo largo del juego, así como los distintos estados de estos. Y para finalizar se explicarán las distintas habilidades con las que contará el usuario durante el desarrollo del juego y el objetivo final del juego.

3.2.1. Pantallas

Knuthians cuenta con una serie de pantallas previas al videojuego en sí para seleccionar el ejercicio a realizar: la pantalla inicial, la pantalla de selección de ejercicio y la pantalla de carga. A continuación se describen en profundidad cada una de ellas.

Pantalla inicial

Es la primera pantalla, en la que se muestran dos botones:

- *Empezar*: este botón permite pasar a la pantalla de selección de ejercicio que se describe a continuación. El usuario podrá notar que éste es el botón a pulsar para jugar porque presenta una ligera animación.
- *Salir* : este botón termina la ejecución de la aplicación.

La figura 3.1 muestra una captura de esta pantalla.

Figura 3.1: Pantalla inicial de Knuthians.

Pantalla de selección de ejercicios

Tras pulsar el botón *Empezar* de la pantalla inicial, se carga esta pantalla. En ella el usuario puede visualizar un listado de todos los ejercicios actualmente almacenados en su carpeta de mapas. Además se encontrará con los siguientes botones:

- *Iniciar juego*: este botón permanece oculto al mostrarse inicialmente la pantalla. Se activa al seleccionar un ejercicio de la lista y pulsar el botón *Cargar XML de la lista*. Al pulsar este botón se pasa a la pantalla de carga donde se crearán los elementos del juego.
- *Cargar XML de la lista*: este botón carga el ejercicio actualmente seleccionado en la lista. Al entrar en la pantalla, el botón presenta una ligera animación y un color distinto para indicar al usuario que este botón es el que se debe pulsar.
- *Volver*: este botón carga la pantalla inicial.

En esta pantalla el usuario debe elegir el ejercicio que desea realizar y cargarlo. Los pasos que debe de seguir para conseguirlo son los siguientes:

- Primero debe seleccionar de la lista el ejercicio que deseé realizar.

- Tras seleccionarlo debe pulsar el botón *Cargar XML de la lista* que presentará una ligera animación y un color azul.
- Tras pulsarlo aparecerá otro botón por encima de éste; el botón será *Iniciar juego*. El usuario deberá pulsarlo para iniciar el juego.

La figura 3.2 muestra una captura de esta pantalla.

Figura 3.2: Pantalla de selección de ejercicios de Knuthians.

Pantalla de carga

Tras pulsar el botón *Iniciar juego* de la pantalla de selección de ejercicios se carga esta pantalla. En ella se ve una esfera girando. Mientras tanto se irán cargando todos los elementos que compongan el mapa seleccionado en la pantalla anterior.

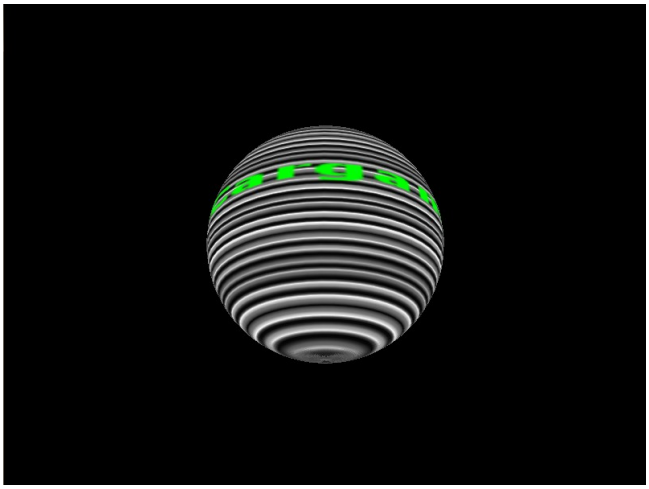

Figura 3.3: Pantalla de carga de Knuthians.

Cuando la carga del ejercicio haya terminado, esta pantalla desaparecerá dando paso a la pantalla de la partida. La figura 3.3 muestra una captura de esta pantalla.

3.2.2. La partida

Una vez elegido y cargado un ejercicio, se mostrará al usuario el mundo de Knuthians, un mundo cuyo objetivo será enseñar cómo se resuelven las gramáticas de atributos de forma entretenida. A continuación se presentarán los elementos que aparecerán a lo largo del mapa, así como sus acciones y efectos, de forma detallada.

El laberinto

El laberinto es el mundo que se ofrece al usuario para que resuelva el ejercicio. Cada ejercicio tiene su propio laberinto, y tiene la misma estructura arbórea que el árbol sintáctico del ejercicio elegido. Esto se hace así para que el alumno asocie el laberinto con el árbol sintáctico. El laberinto está compuesto de los siguientes elementos:

- **Habitaciones**: las habitaciones representan las distintas categorías sintácticas que componen el árbol. Para orientar al usuario por el laberinto, cada habitación tiene por encima de ella una etiqueta en color azul

indicando el nombre de la categoría sintáctica a la que corresponde. Dentro de cada habitación, en el centro, hay una mesa; más adelante explicaremos cual es su función. La figura 3.4 muestra una captura donde se puede ver el interior de una de estas habitaciones.

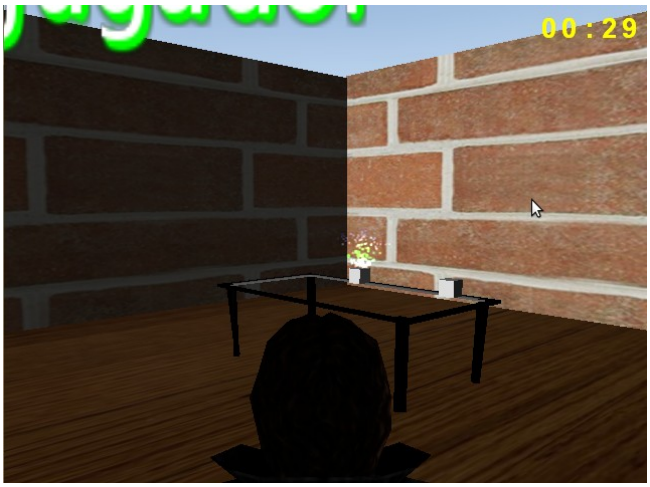

Figura 3.4: Captura de una habitación del laberinto.

- **Pasillos**: son los nexos de unión entre las habitaciones. Representan las ramas de los árboles sintácticos. En ellos, puede haber distintas trampas cuyo objetivo, en la versión actual del juego, es introducir una componente lúdica adicional. No obstante, en versiones futuras podrán incorporar contenidos educativos complementarios (por ejemplo, preguntas sobre la materia de procesadores de lenguaje, que deben contestarse correctamente para que la trampa no reste puntos). La figura 3.5 muestra una captura donde se puede ver el interior de estos pasillos.

Figura 3.5: Captura de un pasillo del laberinto.

- **Mesas**: se encuentran en las habitaciones. Contienen los atributos de cada categoría sintáctica. Su cometido es servir de zona de intercambio de atributos entre el usuario y la categoría sintáctica representada por la habitación. Para interactuar con ella el usuario debe acercar el avatar hasta la mesa. Entonces la vista cambiará a una vista aérea de la mesa. Para abandonar la mesa, el usuario debe pulsar la tecla **E**. La figura 3.6 muestra una captura donde se puede ver una de estas mesas.

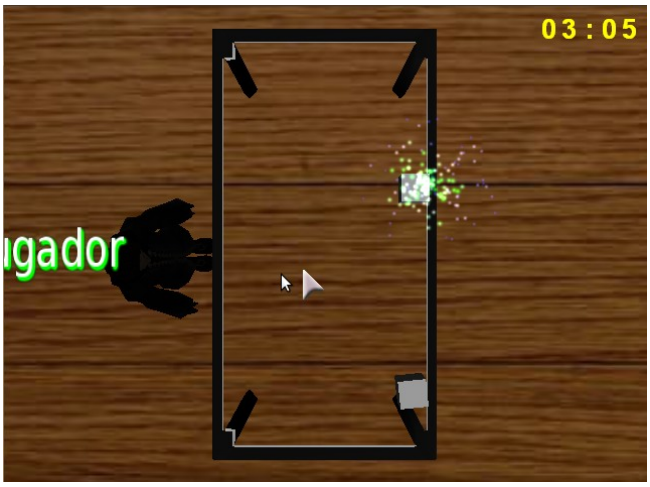

Figura 3.6: Captura de una mesa del laberinto con la vista aérea.

- **Atributos**: se encuentran en las mesas de cada habitación. Son pequeñas cajas de color blanco y representan un atributo de la categoría sintáctica de la habitación donde se encuentran. Estos se pueden encontrar en distintos estados sobre la mesa. Estos estados son:

  - Cuando un atributo no esté sintetizado, no saldrán ningún tipo de partículas de él. Simplemente se observará la caja blanca sin ningún tipo de luz ni efecto.
  - Si del atributo surge un humo gris significará que el último atributo ligado a éste es incorrecto. Esto no significa que todos los atributos correctamente ligados tengan que volver a ligarse. Simplemente nos advierte que el último atributo ligado no es correcto.
  - Cuando un atributo esté sintetizado, de él brotará un chorro de luces multicolores. Estando en este estado, ya se pueden hacer copias de ese atributo al inventario.

  La figura 3.7 muestra capturas de los atributos en sus distintos estados.

Figura 3.7: Capturas de los distintos estados de un atributo.

El usuario puede copiar los atributos activos de la mesa a su inventario. Para ello el avatar deberá estar en contacto con una mesa y seleccionarlo con el botón izquierdo. Además, puede dejar atributos de su inventario en otros atributos que los necesiten para calcular su valor. En este caso, el avatar deberá estar en contacto con la mesa y con el inventario abierto, seleccionar el atributo que desea ligar al atributo de la mesa, pulsar la tecla **Q** y seleccionar con el ratón a qué atributo de la mesa desea ligarlo con el botón izquierdo. Además, se puede obtener información sobre los atributos, como su nombre o su valor, pulsando

con el botón derecho sobre ellos, siempre y cuando el avatar esté en contacto con la mesa.

- **Trampas**: Su cometido es retrasar el objetivo del usuario. Estas trampas no tienen ningún efecto físico directo sobre el avatar del usuario. Sus efectos son otros distintos: pueden hacer que el avatar pierda algunos atributos de su inventario o transportar al avatar a otra posición del laberinto. Para que los efectos de cada trampa tengan efecto, el avatar tiene que chocar contra la trampa. Existen dos tipos de trampas: el bloque que sube y baja en una sección del pasillo impidiendo el paso durante un lapso de tiempo, y el péndulo, que es una esfera atada al techo que oscila a lo ancho del pasillo. Como ya se ha comentado anteriormente, en un futuro estas trampas podrán asociarse con contenidos educativos complementarios. La figura 3.8 muestra unas capturas de las distintas trampas que el usuario puede encontrar en Knuthians.

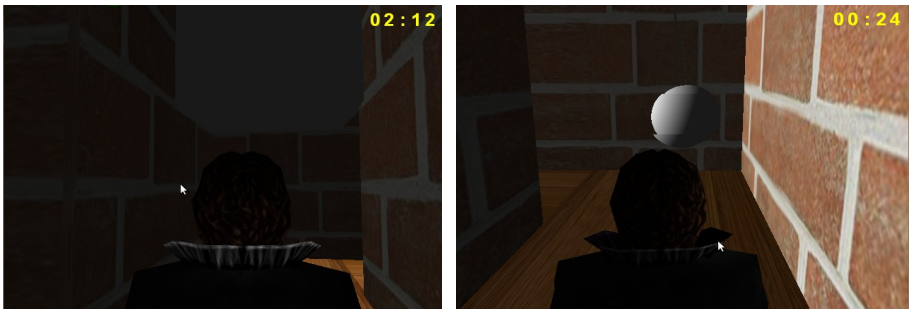

Figura 3.8: Capturas de los dos tipos de trampas de Knuthians.

Además, y como funcionalidad extra, se incluye la vista aérea del laberinto. De esta forma al usuario le puede resultar más sencillo ubicarse dentro del laberinto e incluso entender la forma del laberinto y ver sus similitudes con el árbol sintáctico. La figura 3.9 muestra una captura de la vista aérea.

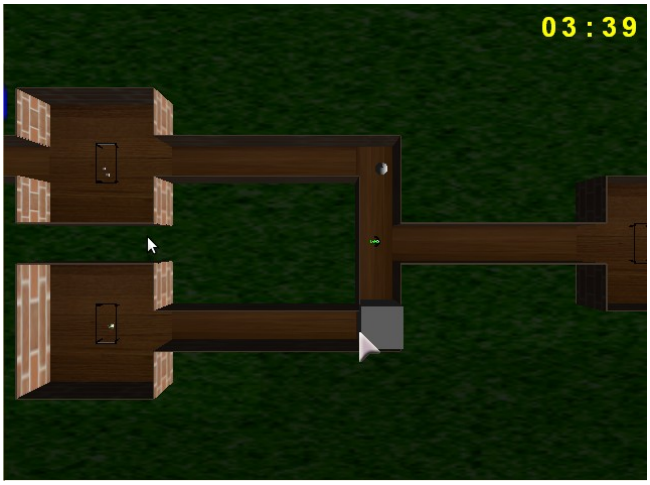

Figura 3.9: Captura de la vista aérea de un laberinto.

El avatar

Es la representación del alumno dentro de Knuthians. Con él podrá moverse por el laberinto e interactuar con el resto de elementos. La figura 3.10 muestra el modelo del avatar. El avatar cuenta con dos cualidades indispensables que el usuario necesitará para resolver el ejercicio cargado:

Figura 3.10: Imagen del avatar que el usuario manejará.

El movimiento

El avatar dispone de capacidad motriz. Cuenta con el típico movimiento de todo juego de plataformas, salvo el salto, ya que no es necesario para nada. El avatar puede moverse adelante con la tecla **W**, atrás **S**, girar a la izquierda **A** y a la derecha **D**. Esto le permite recorrer todo el mundo creado para el ejercicio, y así llegar hasta las distintas habitaciones donde coger los atributos que necesite.

El inventario

Es el lugar donde el avatar carga con los atributos recolectados (más precisamente, con copias de los valores de dichos atributos). El usuario puede abrir y cerrar el inventario pulsando la tecla **I**. Para navegar a través de los distintos atributos que tenga almacenados en ese momento, se usan las teclas **R** y **F**.

Figura 3.11: Imagen que muestra el inventario abierto.

Siempre que el avatar esté rozando una mesa y tenga el inventario abierto, el usuario podrá pulsar la tecla **Q** para dejar al atributo seleccionado dentro del inventario sobre un atributo dentro de la mesa, seleccionándolo con el puntero.

La figura 3.12 muestra un resumen de las teclas utilizadas en el videojuego.

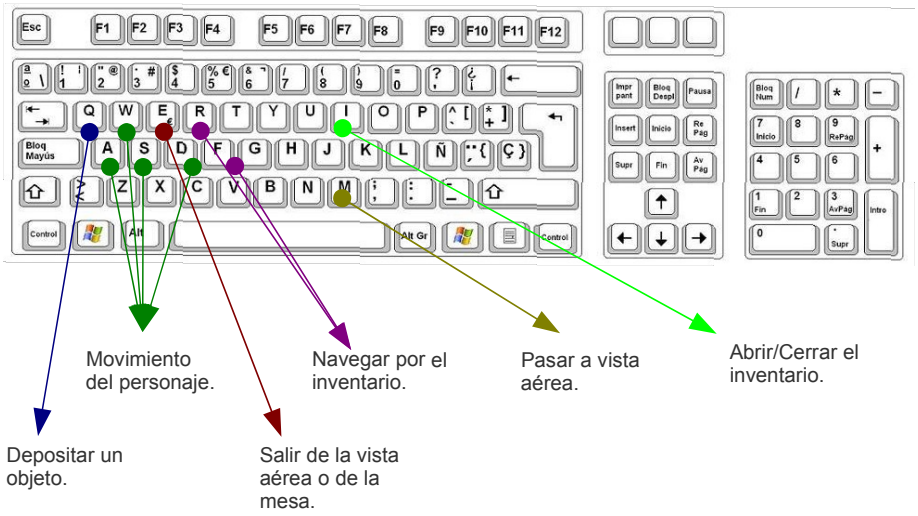

Figura 3.12: Uso del teclado en Knuthians.

Victoria y derrota

Para superar con éxito un laberinto de Knuthians, el usuario debe sintetizar todos los atributos no sintetizados inicialmente de todas las habitaciones del laberinto. Esto se traduce, dentro del videojuego, en conseguir que todas las cajas blancas de todas las habitaciones expulsen un chorro de luz multicolor. Para ello, el usuario deberá recorrer el laberinto recolectando atributos y ligándolos a otros para resolver las dependencias exigidas por la gramática en la que esté basado el ejercicio.

La derrota no existe como tal en Knuthians: un usuario puede intentar resolver un ejercicio durante tanto tiempo como deseé. Si el usuario lo desea puede abandonar una partida cuando quiera, lo cuál se considera una derrota.

Además, para ayudar al usuario a autoevaluarse, Knuthians cuenta con un temporizador que contabiliza los minutos que lleva invertido un usuario

en resolver el laberinto. Así, el usuario puede volver a jugar e intentar batir su propio récord. En el futuro, podrán incluirse mecanismos que permitan fijar la finalización del juego, tales como tiempo límite, o carga vital que disminuye como consecuencia de la caída en trampas o la realización incorrecta de acciones de evaluación.

3.3. Editor de ejercicios

El tutor será el encargado de diseñar los ejercicios que sus alumnos deberán resolver. Para este fin se ha desarrollado una sencilla aplicación para diseñar este tipo de ejercicios, cuyo resultado es un fichero XML que los alumnos podrán cargar, generándose dinámicamente el mapa del juego.

La aplicación ha sido desarrollada en Java, y cuenta con una interfaz estándar Swing. El editor de ejercicios se centra, básicamente, en la edición de árboles sintácticos decorados con atributos, así como de las interdependencias que existen entre dichos atributos. Este editor permitirá añadir los siguientes elementos:

- Nodo no terminal. La figura 3.13 muestra la sucesión de pasos a seguir para añadir un nodo no terminal al ejercicio.

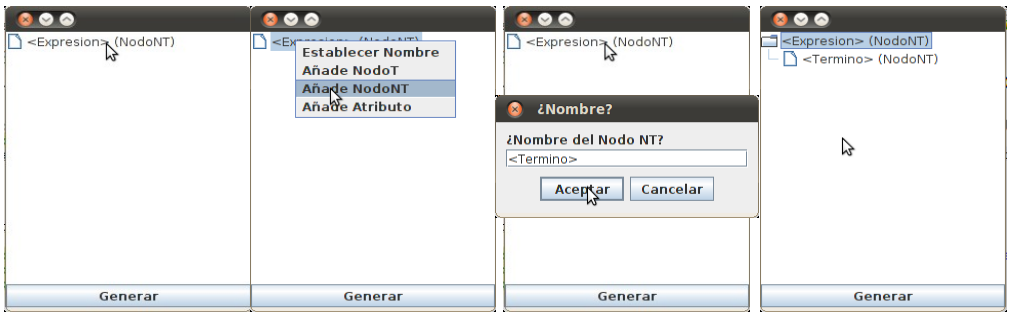

Figura 3.13: Secuencia de acciones para añadir un nodo no terminal en el editor.

- Nodo terminal. La figura 3.14 muestra la sucesión de pasos a seguir para añadir un nodo terminal al ejercicio.

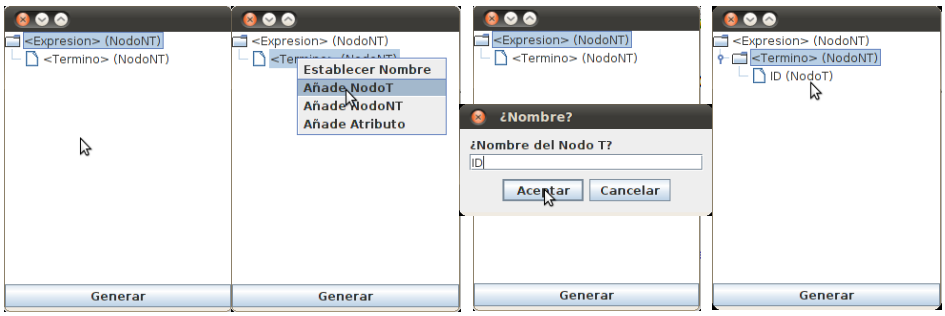

Figura 3.14: Secuencia de acciones para añadir un nodo terminal en el editor.

- Atributo. La figura 3.15 muestra la sucesión de pasos a seguir para añadir un atributo a un nodo del ejercicio.

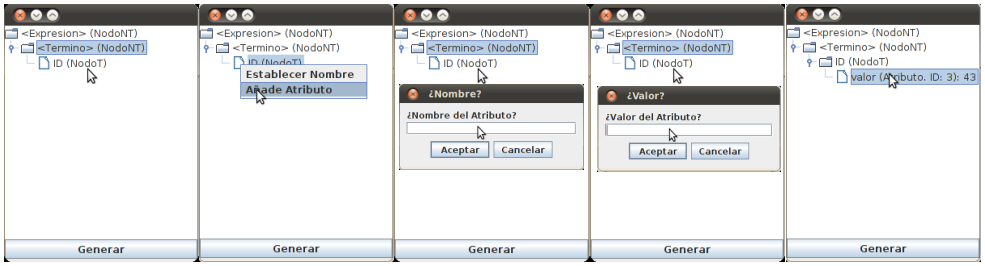

Figura 3.15: Secuencia de acciones para añadir un atributo a un nodo en el editor.

- Dependencias entre Atributos. La figura 3.16 muestra la sucesión de pasos a seguir para añadir una dependencia entre atributos del ejercicio.

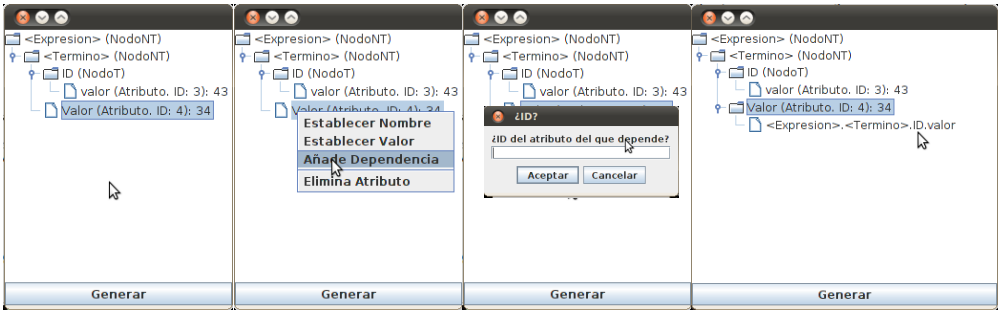

Figura 3.16: Secuencia de acciones para añadir dependencias entre atributos en el editor.

En cualquier momento de la ejecución es posible también cambiar el nombre a los nodos, para prevenir errores de nombrado, así como cambiar el nombre de los atributos y sus valores.

Una vez se haya diseñado el ejercicio utilizando el editor (con estructura de árbol), deberá pulsar en el botón "Generar". Se pedirá, entonces un directorio en el que salvar el archivo, y en él se guardará el fichero XML que codifica el ejercicio. Este fichero XML se cederá a los alumnos, que podrán cargarlo directamente con la interfaz del juego, y podrán empezar a jugar resolviendo el ejercicio de propagación de atributos diseñado. La figura 3.17 muestra un fragmento del código XML generado por el editor de ejercicios.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_18" class="java.beans.XMLDecoder">
 <object id="NodoNT0" class="Prototipo.Modelo.NodoNT">
  <void property="ID">
   <string>0</string>
  </void>
  <void property="atributos">
   <void method="add">
    <object id="Atributo0" class="Prototipo.Modelo.Atributo">
     <void property="nombre">
      <string>pesoh</string>
     </void>
     <void property="pertenece">
      <object idref="NodoNT0"/>
     </void>
    </object>
   </void>
   <void method="add">
    <object class="Prototipo.Modelo.Atributo">
     <void property="dependeDe">
      <void method="add">
       <object idref="Atributo0"/>
      </void>
      <void method="add">
       <object id="Atributo1" class="Prototipo.Modelo.Atributo">
        <void property="dependeDe">
         <void method="add">
          <object id="Atributo2" class="Prototipo.Modelo.Atributo">
           <void property="dependeDe">
            <void method="add">
             <object idref="Atributo0"/>
            </void>
           </void>
           <void property="nombre">
```

Figura 3.17: Fragmento de un XML generado por el editor de ejercicios.

Capítulo 4

Desarrollo e Implementación

4.1. Introducción

Knuthians está construido sobre JMonkey, el motor de videojuegos para Java presentado en las secciones 2.3.2 y 4.5.1. Para servir a nuestro propósito educativo decidimos crear un modelo representativo de los árboles sintácticos para separar el videojuego del árbol sintáctico y así, en un futuro, poder usarlo para construir otros juegos.

En esta sección se describe el desarrollo de Knuthians empezando por la metodología de desarrollo seguida, explicando el modelo usado para describir el árbol sintáctico y, finalmente, describiendo cada una de las partes que componen el videojuego.

4.2. Método de Desarrollo

La primera dificultad de este proyecto residía en que ninguno de los integrantes del grupo había trabajado en el desarrollo de un videojuego anteriormente. Por ello, nos llevo un tiempo acostumbrarnos a utilizar el motor de videojuegos escogido.

Por otro lado, aunque los objetivos del proyecto estuvieron claros desde el principio hacer un videojuego educativo para mostrar el funcionamiento de las gramáticas de atributos presentadas por D.Knuth, el modo de presentarlo se fue perfilando a lo largo del proyecto durante reuniones a lo largo del año.

Por ello, se optó por realizar pequeñas iteraciones en las que se fueron añadiendo funcionalidades y complejidad a un juego muy simple en el que

sólo existía una superficie con una esfera que se podía mover con el teclado. Partiendo de este pequeño ejemplo extraído de la documentación de JMonkey, se fue construyendo el juego añadiéndole funcionalidades como: distintas cámaras, inclusión del motor de física de JMonkey, generación del terreno, uso de luces, una GUI ,etc, hasta llegar a la versión final del videojuego.

Finalmente decidimos crear un generador de ejercicios para el videojuego, para facilitar la labor del docente a la hora de proponer mapas para el videojuego y entregárselos al alumno como ejercicios.

4.3. Arquitectura general de Knuthians

La figura 4.1 muestra de forma simplificada la arquitectura que subyace tras Knuthians.

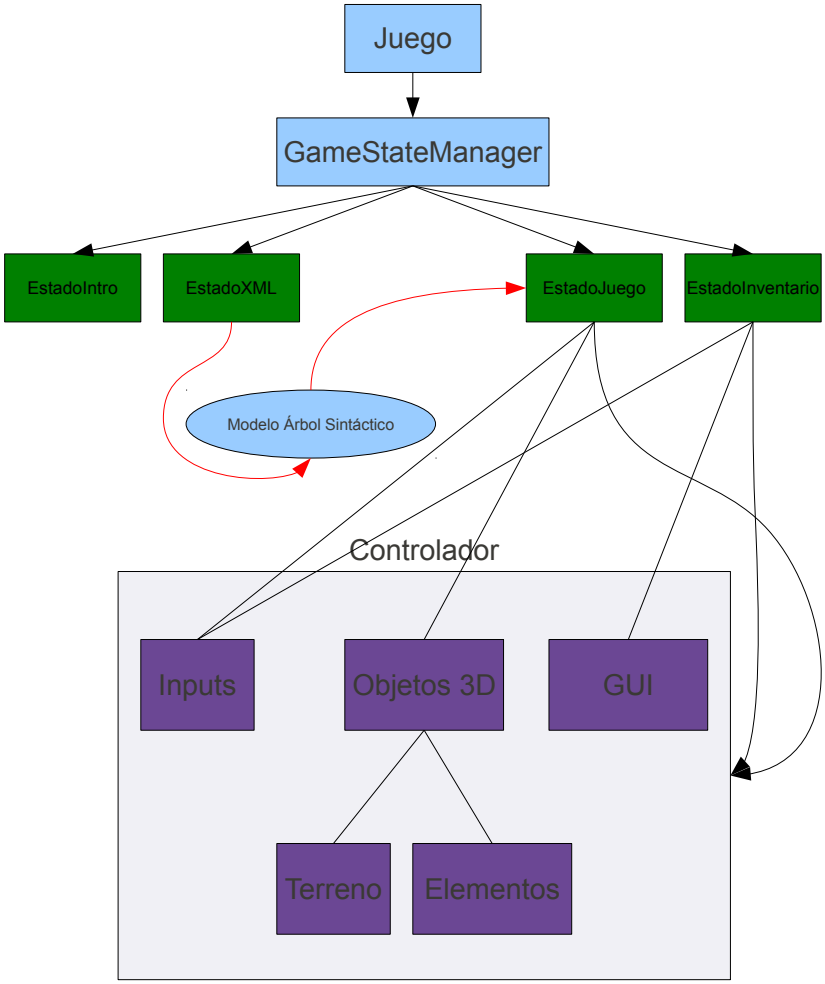

Figura 4.1: Esquema de la arquitectura de Knuthians.

Como se observa en la figura, todo comienza con el bucle principal del videojuego. Dentro de éste se encuentra un gestor de estados de juego que permite cargar, activar y desactivar dichos estados. Estos estados son como hilos dentro del videojuego. En un principio se cargan los estados de introducción y de carga de ejercicios en XML. Una vez el usuario haya seleccionado

un ejercicio, se cargan los estados de videojuego e inventario. El estado del videojuego se encargará de cargar el laberinto y el resto de objetos mediante el procesado del modelo del árbol sintáctico extraído del XML seleccionado por el usuario. Además se cargarán las acciones asociadas a cada entrada del usuario para interactuar con los elementos cargados en ambos estados. Todo este proceso es supervisado por el controlador, donde se almacenará la información clave para gestionar todo el flujo del videojuego. Con todo ello se puede comenzar la partida. En las secciones posteriores se profundizará en cada una de las partes que se describen en este esquema.

4.4. Estructura del Modelo del Árbol Sintáctico

Knuthians propone como ejercicios árboles sintácticos para que el usuario resuelva las dependencias entre atributos y aprenda el funcionamiento de la propagación de atributos. Con el fin de modelar estos árboles se construyó el modelo que se muestra en la figura 4.2. El modelo consta de los siguientes elementos:

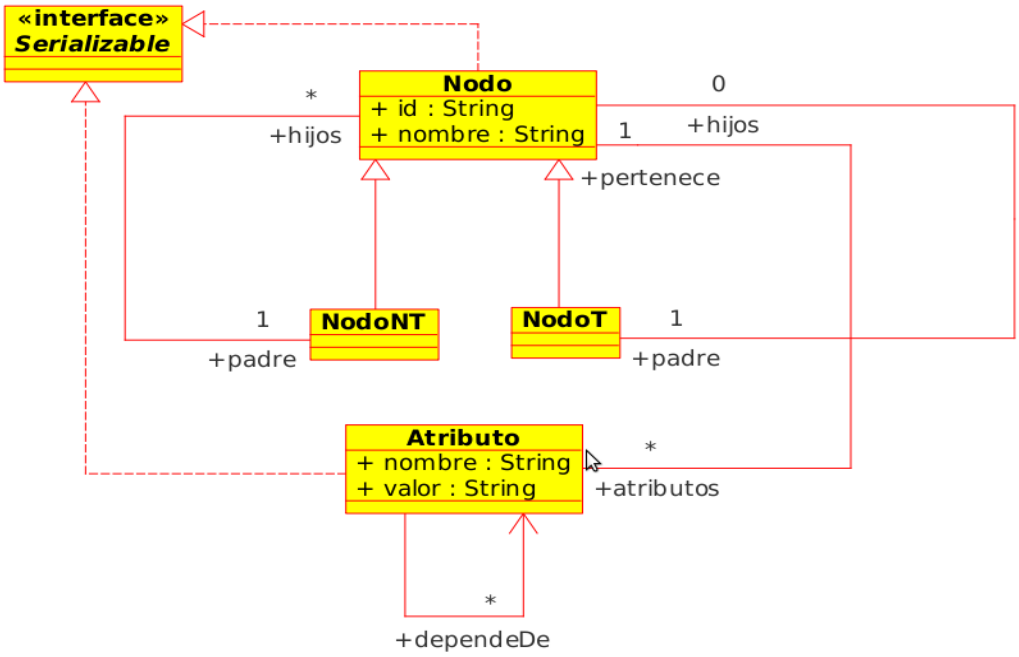

Figura 4.2: Esquema UML del modelo del árbol sintáctico.

- *Nodo* es la clase que representa un nodo genérico del árbol sintáctico. Cuenta con todos los atributos necesarios para definir completamente un nodo. Estos son:

  - **Id** es el identificador único dentro del árbol sintáctico que caracteriza de forma unívoca a cada nodo.
  - **Nombre** es el nombre que mostrará este nodo. Es el nombre de la categoría sintáctica a la que pertenece el nodo del árbol.
  - **Hijos** es la lista de nodos hijos de este nodo.
  - **Padre** es el nodo padre de este nodo.
  - **Atributos** es la lista de atributos asociados a este nodo del árbol sintáctico.

  Además cuenta con métodos accesores y modificadores para todos sus atributos. También cuenta con el método *equals* para compararse con otros nodos.

- *NodoNT* es la clase que representa un nodo del árbol sintáctico que tiene hijos. Esta clase hereda de la clase Nodo y no implementa ningún método adicional ni añade ningún atributo nuevo.
- *NodoT* es la clase que representa un nodo del árbol sintáctico sin hijos, es decir, representa a los tokens del árbol sintáctico. Esta clase hereda de la clase Nodo y al igual que *NodoNT*, no implementa ningún método adicional ni añade ningún atributo nuevo.
- *Atributo* es la clase que representa cada atributo ligado a cada categoría sintáctica del árbol. Cada uno almacenará la siguiente información:
  - **Nombre** es el nombre del atributo. Lo identifica unívocamente dentro de cada nodo.
  - **Valor** contiene el valor de el atributo como una cadena de caracteres.
  - **Pertenece** es un enlace al nodo al que el atributo pertenece.
  - **DependeDe** es una lista que contiene enlaces a todos los atributos que son necesarios para computar el valor de este atributo.

  Además cuenta con métodos accesores y modificadores para todos sus atributos. También cuenta con el método equals para compararse con otros atributos.

Todas las clases anteriores implementan la interfaz *Serializable* (java.io. Serializable). Esto se hace para conseguir la persistencia de los árboles sintácticos construidos con este modelo. Para que el objeto sea serializable, no sólo es necesario que implemente la interfaz *Serializable* sino que tendrá que implementar una constructora por defecto (sin parámetros) y tanto métodos accesores como modificadores. Gracias al *XMLEncoder* y al *XMLDecoder* de Java (*java.beans.XMLEncoder* y *java.beans.XMLDecoder* respectivamente) se almacenan y se recuperan estos árboles en formato XML. Los métodos para cargar y guardar los árboles se encuentran dentro de la clase *Persistencia*(Véase la sección 4.6).

4.5. Estructura del Videojuego

La arquitectura de Knuthians está fuertemente condicionada por el motor de videojuegos usado, JMonkey. Este motor nos ofrece una gran cantidad de

herramientas que permiten construir juegos de forma sencilla para programadores poco experimentados en el desarrollo de videojuegos.

En esta sección se va introducir el uso de JMonkey en el desarrollo de Knuthians, con una breve descripción de su funcionamiento interno y de las herramientas que ofrece al programador, para después explicar de forma detallada la estructura de Knuthians y su funcionamiento.

4.5.1. JMonkey en el desarrollo de Knuthians

Como ya se ha indicado, JMonkey es el motor de videojuegos que se ha usado para implementar Knuthians. Knuthians delega todas las funciones de pintado 3D y gestión de la entrada a JMonkey. Basicamente, JMonkey se encarga de ejecutar el bucle principal del juego. El bucle principal lleva a cabo las siguientes tareas:

- Gestión de la Entrada del Usuario: se hace al principio del bucle para garantizar que la aplicación reaccionará de manera consistente en todo el fotograma.
- Simulación o Actualización de la Lógica del Juego: donde el juego avanza, decidiéndose los comportamientos de los personajes, actualizar el estado de los objetos, lanzar eventos que provocaran otras futuras acciones,etc. Esta parte no la trata JMonkey.
- Simulación Física: se encarga de mover los objetos según la física subyacente y puede encargarse de actualizar los sistemas de partículas o el avance de las animaciones.
- Dibujado del Mundo.

Por otro lado, JMonkey cuenta con una estructura arbórea donde almacenar objetos a la escena. La raíz de este árbol se conoce, dentro de JMonkey, como el rootNode. Es en esta raíz, o a sus hijos, donde se almacenan los objetos de la escena para que sean pintados.

Además, JMonkey gestiona la entrada del usuario mediante el registro de *InputActions* en un manejador de la entrada llamado *InputHandler* junto con la entrada, de teclado o de ratón, a la que asociar esa acción. Este manejador, también permite registrar acciones a otros disparadores, como a eventos producidos por las físicas del videojuego.

Para finalizar, hablaremos de los estados de JMonkey. Estos estados son como hilos que se pueden lanzar simultáneamente dentro de JMonkey y cada uno puede comportarse como un bucle de juego independiente. JMonkey ofrece herramientas para gestionar estos estados, crearlos, activarlos, etc.

4.5.2. Los estados de juego

El videojuego se construye a través de la unión de escenas para poder dar continuidad al juego. De esta manera se evita tener una sola escena para todo el juego que limite la potencia de la aplicación. Para ello, cada pantalla que se presenta (al inicio de la aplicación para poder seleccionar el mapa, iniciar el juego, pantalla de cargando y ya dentro del propio juego el inventario, pop-up de información y el contador de tiempo), corresponde a diferentes estados que se van cargando unos detrás de otros emulando una máquina de estados. Todos estos estados se sitúan dentro del paquete *Estados* y serán ampliados a continuación con más detalle. Cada uno de ellos contiene una constructora a la que se le pasa por parámetro el controlador de la aplicación para poder gestionar todo correctamente. Dentro de la constructora se crea un *PrincipalGUI* que contiene todo los componentes que pueden hacer falta para gestionar la GUI y se crea el manejador del ratón. Por otro lado, se llama al método *buildUI* que se encarga de ejecutar los componentes que se quieren utilizar en dicha escena.

Una vez creados todos los estados necesarios hay que gestionarlos para que se muestren en el orden correcto. Cada vez que se crea un nuevo estado hay que insertarlo en la cola de JMonkey de escenas con el siguiente comando: *GameStateManager.getInstance().attachChild(estIntro)*. Una vez creado se puede llamar en cualquier momento para que se muestre dicho estado con el comando *GameStateManager.getInstance().activateChildNamed(estIntro)*. Por último, para hacer desaparecer un estado que representa una escena, basta con llamar al comando *GameStateManager.getInstance(). deactivate-ChildNamed(estIntro)*.

La figura 4.3 muestra el diagrama UML donde se muestran las clases que componen el paquete *Estados*. A continuación se explican las diferentes clases de las que se compone el paquete *Estados*.

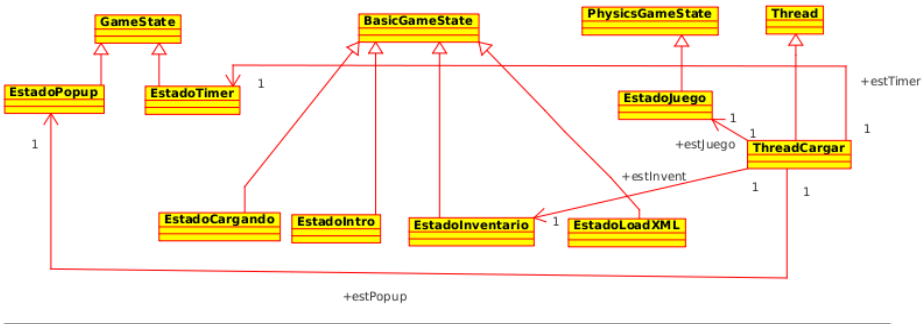

Figura 4.3: Esquema UML de las clases del paquete *Estados*.

ConfiguradorJuego

En esta clase se tienen tres métodos estáticos para poder tener las diferentes configuraciones de manera clara y ordenada dependiendo del modo que se seleccione:

- *modo mapa*: configuración que se aplica cuando el usuario decide utilizar el modo mapa, vista aérea de todo el árbol sintáctico.
- *modo jugador*: configuración que se aplica al iniciar el juego y cuando el usuario decide volver al modo jugador, vista detrás del avatar.
- *modo mesa*: configuración que se aplica cuando el usuario se acerca a la mesa con el avatar, vista aérea de toda la mesa.

EstadoCargando

Esta clase se encarga de crear la pantalla de espera del inicio mientras se espera a que termine la carga de todos los elementos del juego. En dicha pantalla aparece una esfera girando, con una etiqueta en ella mostrando el mensaje de *cargando*.

EstadoIntro

Clase que se encarga de crear la primera pantalla en la que el usuario puede interactuar. En ella se crean dos botones: uno para seguir el proceso de escenas hasta llegar al juego y otro para salir de la aplicación. Cabe

destacar que para guiar al usuario, se muestra el botón de *Empezar* con un ligero movimiento y cambiando el color para que sea más intuitivo, ya que los movimientos recrean una secuencia lógica para llegar a iniciar el juego. Por último se cargan todos los elementos necesarios para tener disponible el ratón y poder interactuar sobre los botones de dicho estado.

EstadoInventario

Clase encargada de crear el inventario. Dicho inventario consta de una lista a la cual se le ha quitado el fondo para dar una mayor integración al inventario dentro del videojuego y también se le ha suprimido el scroll, ya que para moverse a través de la lista se utiliza el teclado. Mencionar que esta clase posee un método llamado *rellenarLista* para recorrer el ArrayList e ir imprimiendo en la lista física toda la información que se tiene hasta el momento.

EstadoJuego

Esta clase implementa el estado más importante del juego, ya que en este estado se inicializa todas las partes de las que se compone el juego. Este estado entra en acción nada más terminar de cargarse todo los componentes, al finalizar el *estadoCargando* y es el encargado de la estructura general del videojuego. A continuación se mostrarán algunos de los métodos de esta clase para ir viendo el proceso de cómo se cargan las diferentes partes de las que se compone el juego en sí. El método principal es *initGameState*. A éste se le pasa por parámetro un Nodo que indica el modelo de mapa que se tiene que cargar, previamente elegido por el usuario. Dentro de este método se inicializa el Timer del juego de JMonkey y se prepara todas las partes de las que se compone el juego. Algunas métodos destacables en el *initGameState* son:

- *initTerrain*: Se encarga de construir el terreno de acuerdo con el modelo elegido anteriormente por el usuario al iniciar la aplicación.
- *initPlayer*: Se encarga de generar el avatar. Para ello se carga el modelo 3D, se añaden las físicas, se inserta en el terreno y por último se añade al controlador para tener acceso a él en cualquier momento.
- *initTraps*: Se encarga de inicializar las trampas en el juego. Para ello se da la posición donde se quiere insertar dicha trampa, el tipo de trampa

y el efecto que se quiere conseguir al hacerse efectiva dicha trampa. Por último se añade las trampas al controlador para poder acceder a ellas en cualquier momento del juego.

- *initLights*: Se encarga de configurar e inicializar adecuadamente la luz direccional y la luz puntual del juego para que se vea perfectamente las escenas a lo largo del juego.

- *initCameras*: Se encarga de inicializar la cámara. En este caso pone por defecto la cámara en modo jugador para que aparezca así en el inicio del juego y por último se añade dicha cámara al controlador para poder tener acceso a ella.

- *initInput*: Se encarga de crear y configurar adecuadamente todas las entradas posibles para tenerlas registradas y saber qué ejecutar cuando una de esas entradas entre en acción en cualquier parte del juego. Algunas de estas son:

  - *playerInput* relacionadas con el usuario sobre la entrada por teclado para dirigir al avatar.
  - *camPlayerInput*, *camMapInput*, *tableInput* y *inventaryInput*, todas ellas relacionadas con los cambios de cámara. Para cada entrada se activa o desactivan ciertas teclas del teclado, para restringir y así utilizar las funcionalidades adecuadas cuando corresponda.
  - *initMouse*. En esta entrada se crea el ratón y se añade al controlador para poder acceder al ratón.

EstadoLoadXML

Clase que se encarga de crear la selección del archivo XML para poder cargar correctamente el mapa del juego. Este estado consta de tres botones. Uno de ellos, *Iniciar Juego*, estará oculto hasta que se haya realizado todo el proceso de carga de manera correcta. En este momento se hará visible, tendrá animación y cambiará de color. El segundo botón, *Cargar XML de la lista*, aparece inicialmente animado y de otro color, y el último botón, *Volver*, no varía. Además de los tres botones, contiene una lista en la que se carga directamente el directorio completo donde estén almacenados todos los XML. Por último, este estado posee una etiqueta de ayuda en la parte inferior que nos va indicando en todo momento los pasos que debemos de

seguir, y cuando seleccionamos un archivo nos muestra su nombre en dicha etiqueta para saber cuál tenemos seleccionado en ese momento. Por otro lado, destacar la carga del manejador del ratón para poder interactuar en esta pantalla y poder seleccionar los botones y elementos correspondientes.

EstadoPopUp

Clase encargada de crear un pop-up. Este se deja creado al inicio, y cuando se requiera sólo hay que llamarlo con las instrucciones que se mencionaban en el inicio de esta sección sobre cómo cambiar de escena. Al estar añadido al controlador, se puede modificar el texto de dicho pop-up para poder poner el que interese en ese momento concreto. Destacar que hay que tener cuidado con el orden en que se presentan las cosas cuando se utilice el pop-up, ya que si no se puede superponer y no verse el pop-up por quedar escondido entre los componentes de la escena en la que se esté en ese momento.

EstadoTimer

Esta clase es la encargada de crear el reloj que se muestra durante el juego. Este reloj está realizado mediante una etiqueta para poder mostrar los minutos y segundos, que se van actualizando cada segundo. Para poder actualizarlo de la manera mencionada, el reloj está construido mediante un thread que se dispara al inicio del juego y mantiene actualizado el reloj en todo momento. De esta parte se encarga el método *run*, que implementa la clase de la que estamos hablando. Al llegar al final del juego el reloj se para. Así se puede ver el tiempo empleado en resolver el problema. También aparece una nueva etiqueta mostrando el final del juego.

ThreadCargar

Esta clase se encarga de crear el *estadoJuego* mencionado anteriormente. Esta clase permite la utilización de hilos para poder cargar todas las partes del juego y a la vez poder mostrar un *estadoCargando*. Con ello se consigue que todo el proceso vaya más fluido y el usuario apenas perciba el tiempo de espera mientras todo está siendo creado y cargado.

4.5.3. El terreno del juego

En esta sección se abordarán los distintos elementos gráficos que componen el terreno del videojuego. Todos los objetos descritos se encuentran en el paquete *Terreno*. De cada objeto describiremos sus atributos, su utilidad y sus métodos, haciendo especial hincapié en aquellos métodos más importantes. La figura 4.4 muestra el diagrama UML de estas clases.

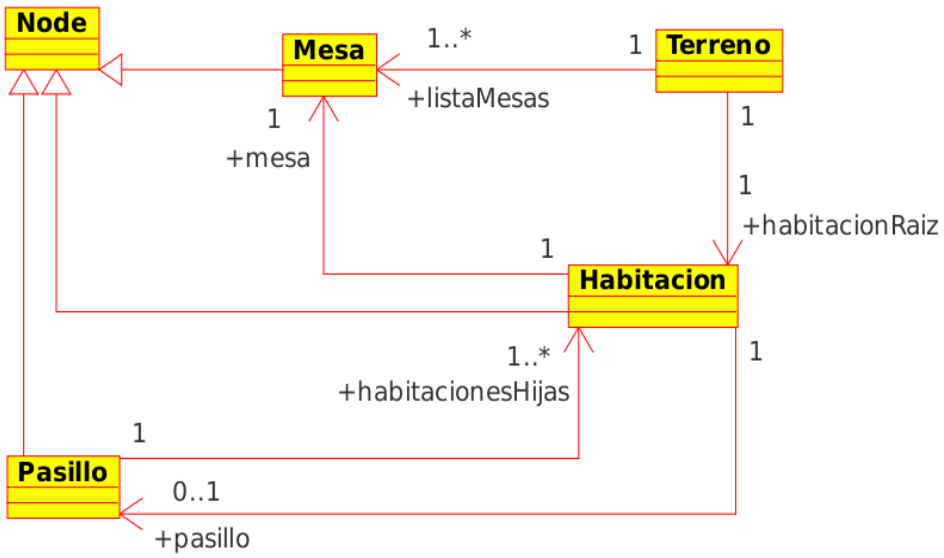

Figura 4.4: Esquema UML de las clases del paquete *Terreno*.

Habitacion

Esta clase es la que se usa para crear las habitaciones. La clase extiende la clase *Node* de JMonkey para que pueda ser incluida dentro del árbol que configura la escena. La constructora tiene los siguientes parámetros de entrada:

- **id** es un String que identificará a la habitación. Este String se corresponde con el id de la clase Nodo del modelo del árbol sintáctico.
- **nombre** es un String que contiene el nombre que mostrará ésta. Este String se corresponde con el nombre de la clase Nodo del modelo del

árbol sintáctico.

- **modo** es un entero que nos indica qué tipo de habitación será la que se construya. Esto afecta a la arquitectura de la habitación. Los posibles valores que puede tomar están registrados dentro de esta clase como constantes y son las siguientes:
  - *INI* nos indica que la habitación será la habitación inicial. Tendrá una sola puerta en la pared sur.
  - *INTER* nos indica que la habitación será una habitación intermedia. Tendrá una puerta en la pared sur y otra en la pared norte.
  - *FIN* nos indica que la habitación no tendrá salida a otro pasillo. Solo tendrá una puerta en la pared norte de la habitación.
- **tamX, tamY** y **tamZ** son los tamaños que tendrá la habitación en cada uno de sus ejes. El **anchoPuerta** es el tamaño que ocuparan las puertas en la pared.

Todos los parámetros de entrada se registran como atributos de la clase, menos el **modo** y el **anchoPuerta**. Además, se almacenan las paredes y el suelo creados para posteriormente tratarlos en otros métodos.

La clase también contiene métodos accesores para los atributos **tamX, tamY** y **tamZ**. Además la clase reescribe el método *draw* para poder añadir la textura a las paredes y el suelo. Esta práctica es muy habitual en muchos objetos de la escena. Otro método importante dentro de esta clase es el método *putTable*, que recibe como parámetro un objeto de la clase *Mesa* y lo sitúa en el centro de la habitación. Más adelante se describen la clase *Mesa* que se encuentra dentro del mismo paquete que esta clase.

Pasillo

Esta clase es la que se usa para crear los pasillos que unirán cada habitación. Esta clase extiende la clase *Node* de JMonkey. Los atributos de la clase son los siguientes:

- **ramas** es un *ArrayList* que almacena el objeto 3D que hará del suelo de cada una de las ramas que llevarán a cada habitación hija.

- **pDramas** y **pIramas** son unos *ArrayList* que almacenan los objetos 3D que harán de las paredes derecha e izquierda, respectivamente, de cada rama.
- **nexo** es un objeto 3D que hará del suelo del pasillo que sale de la habitación padre.
- **pDnexo** y **pInexo** son unos objetos 3D que harán de las paredes derecha e izquierda, respectivamente, del pasillo que sale de la habitación padre.
- **bus** es un objeto 3D que hará del suelo del pasillo transversal que unirá el nexo con las ramas.
- **pIbus** , **pDbus**, **pTIbus** y **pTDbus** son los objetos 3D que harán de paredes izquierda, derecha, pared norte izquierda y pared norte derecha, respectivamente, del pasillo transversal que unirá el nexo con las ramas.
- **pBBus** es un *ArrayList* que almacena los objetos 3D que harán de las paredes sur del bus.
- **tamY** es un número en coma flotante que almacena la altura de las paredes del pasillo.

Además cuenta con constantes que especifican el largo de cada rama y el ancho del pasillo.

La única constructora de la clase tiene como parámetros: el nombre del pasillo, un entero que indica el número de ramas que conectará el pasillo, la altura de las paredes del pasillo y un número en coma flotante que indica la separación entre las ramas. La constructora crea el nexo con sus paredes. Después crea el bus en función del número de ramas y la separación entre las ramas introducidos en la constructora. Especial atención requiere la creación de las paredes sur del bus, ya que su tamaño y número varia en función del número de ramas y del espacio entre cada rama. Por último, se crea cada rama con sus paredes.

Esta clase también cuenta con una función accesora a su atributo **tamY**. Además reescribe el método *draw* de la clase *Node* de JMonkey para incluir las texturas necesarias a cada objeto 3D creado.

Mesa

Esta clase es la que se usa para crear una mesa. Además, como las mesas sirven como área de intercambio de atributos que el usuario puede usar a través del avatar, cuenta con los métodos necesarios para acceder a los atributos y gestionarlos. Los atributos con los que cuenta la clase son los siguientes:

- **listaDeAtributos** es un *ArrayList* que contiene todos los atributos almacenados en esa mesa.
- **fisicas** es un nodo de físicas estático proporcionado por JMonkey (*StaticPhysicsNode*). Se utiliza para proporcionarle propiedades físicas a la mesa y para poder registrar un evento de colisión.
- **collisionEventHandler** es un botón sintético que emula a una entrada del usuario. Con esto, hacemos que la colisión entre la física de la mesa con cualquier otro objeto con físicas se trate como otra entrada del usuario.
- **ctr** es un enlace al controlador del videojuego para poder acceder a información necesaria para gestionar los atributos que se encuentran en la mesa.

La única constructora de la clase tiene como parámetros: un nombre para la mesa, el enlace al controlador del videojuego y el nodo de físicas. La constructora carga el modelo 3D de la mesa, inicializa el *ArrayList* y registra la colisión con la física de la mesa como una entrada del usuario con un *InputAction* creado que gestiona la colisión.

Además cuenta con métodos para gestionar los atributos situados en la mesa. Estos métodos son:

- **putAttribute** es un método que sitúa el atributo pasado como parámetro en una posición aleatoria de la mesa, actualizando el atributo **listaDeAtributos**.
- **linkAttribute** es un método que lleva a cabo la resolución de una dependencia entre dos atributos de entrada. Si el atributo a añadir al atributo objetivo no es necesario, se activarán unas partículas de humo que brotarán del atributo.

- **initialCheck** es un método que recorre todos los atributos de la mesa y comprueba su estado para activar aquellos atributos que no tengan dependencias.

Terreno

Es la clase más importante de este paquete. Se encarga de construir el laberinto que presentará la partida a partir del modelo cargado desde el XML del ejercicio. Además, permite situar al jugador dentro del laberinto en cualquier posición. Los atributos de la clase son los siguientes:

- **name** es el nombre del terreno.
- **posIni** es un vector de tres números en coma flotante que define la posición inicial en la que aparecerá el jugador.
- **sceneNode** es el nodo de la clase de JMonkey *Node* donde se añadirán todos los objetos generados para construir el laberinto.
- **fisicas** es el espacio de físicas (*PhysicsSpace*) proporcionado por JMonkey para crear nodos con propiedades físicas.
- **input** es un manejador de entrada de usuario para registrar los botones sintéticos creados por cada mesa.
- **ctr** es un enlace al controlador del videojuego.
- **mesas** es un *ArrayList* donde se almacenan todas las mesas creadas en el laberinto.

La constructora inicializa los atributos de la clase usando los parámetros de entrada. Estos son: el nombre, la posición inicial donde aparecerá el jugador, el nodo raíz de la escena, el espacio de físicas y el enlace al controlador. Los métodos más importantes son **generateMap** y **putTables** que se pasan a explicar a continuación.

El método **generateMap** recorre un árbol sintáctico modelado con la estructura presentada en el apartado anterior utilizándolo de forma recursiva. Mientras lo recorre, va generando las habitaciones, una por cada nodo del árbol sintáctico, pero no genera los pasillos. Los pasillos necesitan saber el espacio entre las ramas, que dependerá del número de hojas que cuelguen de este pasillo. Este dato se calcula al final de la recursión, así que los pasillos se

construyen usando el valor que devuelve la llamada a la recursión siguiente. Esto se hace para asegurarse que ningún pasillo o habitación se solapen entre sí, asegurándose de que los pasillos más cercanos a la habitación raíz tengan más espacio entre las ramas. Una vez generado todo el laberinto, se crea las física que envolverá a las paredes, el suelo de las habitaciones y los pasillos. Tras generar el laberinto, se pasa a crear las mesas y situarlas en cada habitación. Esta tarea se lleva acabo gracias al método *putTables* que recorre el árbol de la escena creado por *generateMap* y sitúa en cada habitación una mesa y activa su "botón simulado" para realizar la acción registrada cuando otro objeto con físicas colisione con la física de la mesa. Además, este método nos devuelve una lista con los atributos que no están sintetizados dentro del árbol sintáctico para poder registrarlos en el controlador. La figura 4.5 muestra de forma esquemática cómo se recorre el árbol sintáctico y como se va construyendo el laberinto a la vez.

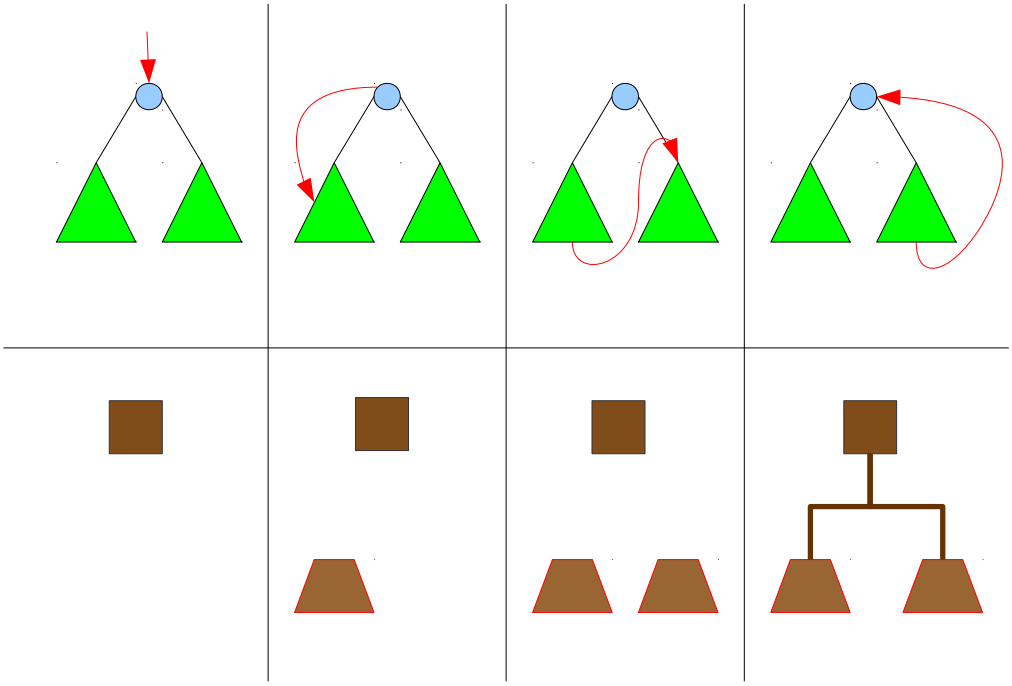

Figura 4.5: Esquema de la construcción del laberinto a partir del árbol sintáctico

Además, esta clase cuenta con métodos para situar el avatar del jugador dentro del laberinto y teletransportarlo, también.

4.5.4. Los elementos del juego

En esta sección se abordarán los distintos elementos gráficos que se encontrarán dentro del terreno del videojuego. Todos los objetos descritos se encuentran en el paquete *Elementos*. De cada objeto describiremos sus atributos, su utilidad y sus métodos, haciendo especial hincapié en aquellos métodos más importantes. Además, dentro de este paquete se encuentran otros dos paquetes que contienen clases para las distintas trampas y el otro para sus efectos. La figura 4.6 muestra el diagrama UML de las siguientes clases.

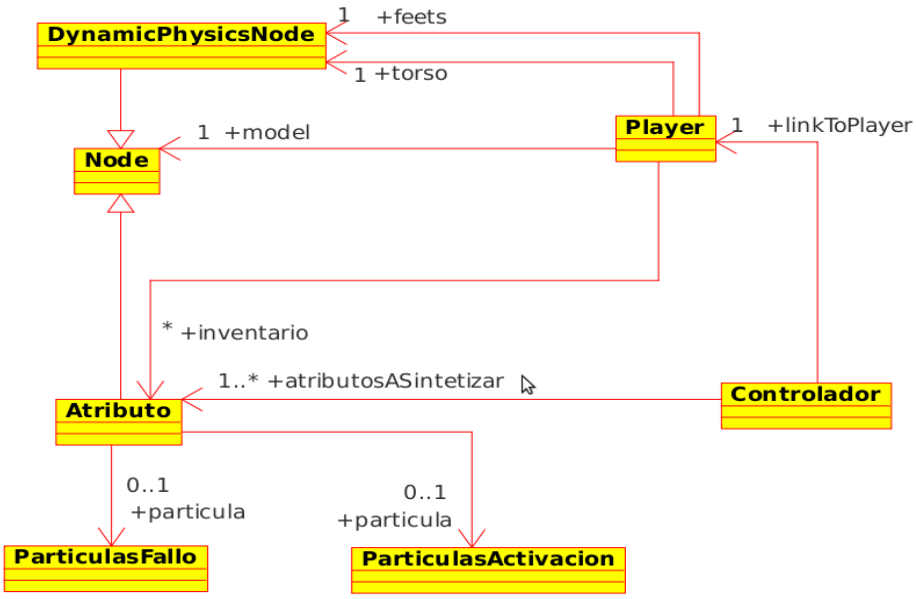

Figura 4.6: Diagrama UML de las clases que componen el paquete *Elementos*.

Player

La clase *Player* representa el avatar que el usuario manejará a lo largo del juego. Para ello produce el modelo 3D del avatar y también los elementos de

físicas necesarias para poder mover el avatar. Proporciona los métodos necesarios para mover el avatar a lo largo del terreno mediante sus físicas. Además contiene las herramientas necesarias para gestionar el inventario donde almacenará las copias de atributos que irá haciendo. Los atributos que contiene esta clase son:

- **inventario** es un *ArrayList* donde se almacenarán las copias de atributos que haga el usuario.
- **charNode** es un *Node* de JMonkey donde se añadirán el resto de nodos que componen el avatar.
- **feetNode** es un *DynamicPhysicsNode* de JMonkey. Esto permitirá el movimiento al avatar que manejará el usuario.
- **physNode** es un *DynamicPhysicsNode* de JMonkey. Éste objeto envuelve al modelo que representa al avatar para poder detectar colisiones con el modelo.
- **model** es un *Node* de JMonkey donde se almacenará el modelo 3D del avatar.
- **onGround** es un booleano que indica si el avatar esta posado sobre el suelo o no.

La constructora tiene como parámetros: un String con el nombre del jugador, el modelo 3D del avatar y un espacio de físicas para generar los nodos de físicas dinámicos necesarios para el avatar. La constructora carga el modelo en el atributo **model**. Antes de continuar con la construcción del avatar se comprueba que el modelo cargado incluya las animaciones de movimiento. Después se crean las físicas que regirán el movimiento y la detección de colisiones del avatar. En este caso se ha recurrido a seguir un esquema de físicas muy usual en juegos en tercera persona. La figura 4.7 representa de forma esquemática la estructura de las físicas para el avatar. El bloque verde se corresponde con el atributo **physNode** y se usa para detectar las colisiones con el torso del avatar. El bloque azul se corresponde con el atributo **feetNode** se corresponde con los pies del avatar, será el encargado de dotar de movimiento al avatar.

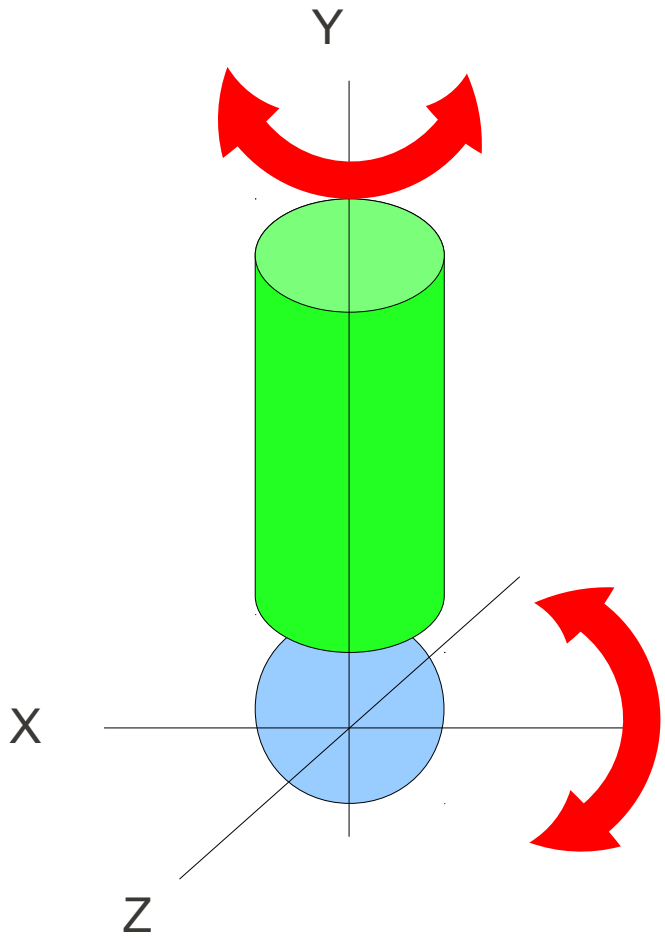

Figura 4.7: Esquema de la organización de las físicas para el avatar.

Para que el avatar se pueda mover hacia delante o hacia atrás sólo hay que aplicar una velocidad angular a la esfera que representa sus pies en el eje X. El signo de esa velocidad indicará en que sentido se mueve el avatar. Para rotar el avatar sólo es necesario aplicar a los dos bloques de físicas, la esfera y el cilindro, una velocidad angular en sus ejes Y.

Además, la clase contiene métodos para encapsular los posibles movimientos del avatar, giro y avance. También tiene métodos para añadir y quitar atributos al inventario del avatar.

Atributo

Esta clase representa los atributos que se encontrarán en las mesas de las habitaciones. Es la representación de los atributos que se encuentran en el modelo del árbol sintáctico, por lo que entre sus atributos (como clase) se puede encontrar atributos que contienen los atributos del modelo del árbol sintáctico. Esta clase hereda de la clase *Node* de JMonkey. Los atributos de esta clase son:

- **origen** es un String donde se almacenará el **id** de la habitación a la que pertenece el atributo.
- **origenNombre** es un String donde se almacenará el **nombre** de la habitación a la que pertenece.
- **activo** es un booleano que indica si el atributo está ya sintetizado o no.
- **dependencias** es un *ArrayList* de String que contendrá las dependencias de este atributo. Cada dependencia es un String que tiene la siguiente forma : "id de la habitación a la que pertenece el atributo del que depende"."nombre del atributo del que depende".
- **cumplidas** es un *ArrayList* de booleanos que tiene el mismo número de elementos que **dependencias**. Nos indica qué dependencias se han cumplido y cuales no.

La constructora crea la caja en 3D y la asocia al objeto. Después inicializa los *ArrayList* e introduce los parámetros de entrada restantes de la constructora con el atributo adecuado.

Aparte de las accesoras y modificadoras para los atributos de la clase cuenta con los siguientes métodos para gestionar las dependencias del atributo:

- **addDependencia** que añade al *ArrayList* de **dependencias** la dependencia. El String introducido debe tener la forma explicada anteriormente. También añade al *ArrayList* de **cumplidas** un falso en la misma posición.

- **isActivate** que comprueba todos los booleanos de **cumplidas** para saber si ya se puede marcar el atributo **activo** del atributo.
- **esNecesario** que tiene como entrada un String de dependencia. Comprueba si la dependencia está registrada en **dependencias**. Si la dependencia esta registrada en **dependencias** y no ha sido procesada ya, se invoca al método **isActivate** para comprobar si era la última dependencia por comprobar y actualizar así el booleano **activo**.
- **activate** es el método que se encarga de mostrar visualmente la activación de un atributo, añadiéndole una partícula de activación (clase *ParticulaActivacion* de este mismo paquete) que se explicará más adelante.

Partículas

En Knuthians existen dos tipos de partículas: las partículas de fallo, que aparecen cuando se añade un atributo a otro como dependencia y no era necesario, y las partículas de activación, que aparecen cuando un atributo está sintetizado. Cada una de ellas cuenta con una clase distinta, pero muy parecidas, que son *ParticulasActivacion* y *ParticulasFallo*. Contienen un atributo de tipo *ParticleSystem* donde se almacenará la partícula, y ambas tienen la constructora donde se configura la partícula y otro método para obtener la partícula configurada en la constructora.

Dependiendo de la partícula, se configurará de un modo u otro.

TextLabel2D

Esta clase sirve para crear etiquetas en 2D con el texto y color que se quiera. La constructora tiene como parámetros el texto a introducir. Ofrece gran cantidad de posibilidades de configuración mediante distintos métodos. Se puede cambiar el color, la fuente, la sombra, etc.

Trampas

Este paquete contiene una clase abstracta, *Trampa*, que define los métodos y atributos necesarios para cualquier trampa y que deben extender todas las trampas. Los atributos son:

- **origin** es un vector (*Vector3f*) que indica la posición inicial de la trampa dentro del laberinto.
- **physics** es un espacio de físicas (*PhysicsSpace*) para que en las trampas que extiendan de esta clase se puedan crear los elementos físicos que se deseen.
- **effect** es el efecto que se asocia a la trampa. Son objetos que implementan la interfaz *IEffect* que se explicará más adelante.
- **input** es un manejador de entrada de usuario para registrar los botones simulados creados para reaccionar ante las colisiones de las físicas de los objetos creados para la trampa.

La constructora tiene como parámetros los mismos que los atributos, los cuales asigna a estos dentro de la constructora. Cuenta también con el método privado *performAction* que invoca al método *performAction* de **effect**. Además propone dos métodos abstractos que se deben implementar en las trampas concretas. Estos métodos son:

- **update** que se encargará de actualizar en cada vuelta del bucle la dinámica de las físicas y la lógica de la trampa, en caso de que el programador lo considere necesario.
- **putTrampa** que añadirá los objetos 3D necesarios al nodo pasado por parámetro para construir la trampa.

Usando esta clase abstracta se han construido las dos trampas que se pueden encontrar en el videojuego dentro de este paquete.

Effects

Dentro de este paquete se encuentra una interfaz que deben implementar todas las trampas. Esta interfaz *IEffect*, contiene un método llamado *performAction* que será el encargado de cambiar el estado de los elementos que el programador del efecto considere necesarios.

Para Knutians se han programado dos efectos distintos. *LoseAttr* es una clase que implementa la interfaz *IEffect* y en su método *performAction* elimina el último elemento añadido al inventario del avatar. *Teletransporter*, al igual que *LoseAttr*, implementa la interfaz *IEffect*. En este caso el efecto programado en el método de la interfaz teletransporta al avatar del usuario a

una posición dada a la clase como parámetro en la constructora. Un aspecto importante de estos dos efectos programados es que se apoyan en el controlador para poder consultar y modificar la información necesaria para llevar a cabo el efecto.

La figura 4.8 muestra el diagrama UML de las clases que en encuentran en los paquetes *Effects* y *Trampas*.

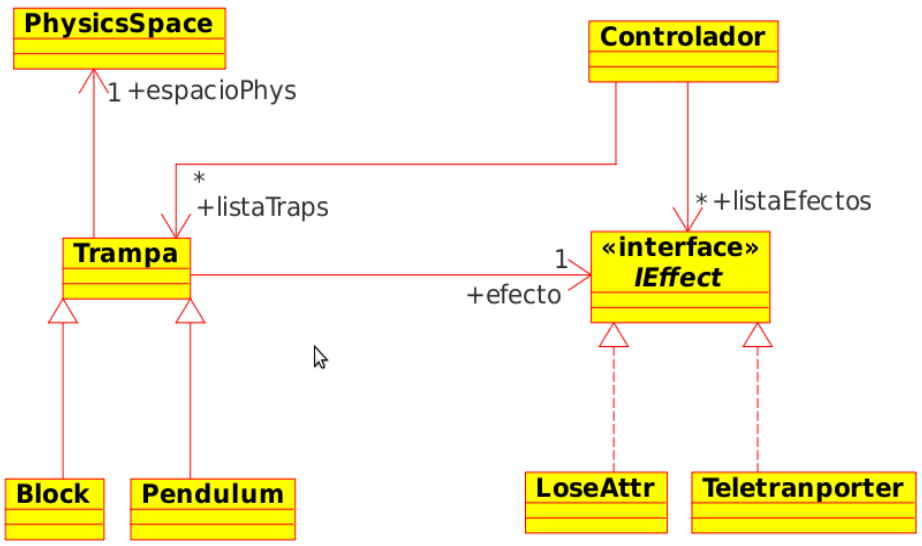

Figura 4.8: Diagrama UML de las clases que componen los paquetes *Effects* y *Trampas*.

4.5.5. Las cámaras

En esta sección se abordará el uso de la cámara dentro de Knuthians, así como las clases necesarias para crearla y usarla. Todas las clases descritas en esta sección se encuentran dentro del paquete *Camaras*. En este paquete se encuentra una clase que define y configura la cámara usada. Además se encuentran tres clases que heredan de la clase *MouseInputAction* de la JMonkey. Estas acciones se asocian directamente a las acciones que procedan del ratón. Estas tres acciones representan las tres vistas distintas que el usuario puede usar a lo largo del juego.

A continuación se va a describir la clase *Camara* donde se crea la cámara. Para finalizar se explicará brevemente cada una de las acciones que manejarán estas cámaras para producir las tres vistas distintas que aparecen en el videojuego.

La figura 4.9 muestra el diagrama UML de las clases que componen el paquete *Camaras*.

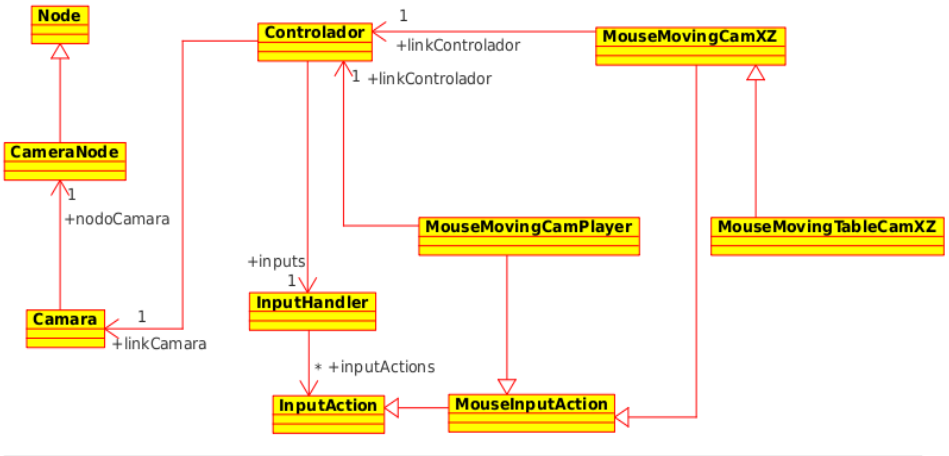

Figura 4.9: Diagrama UML de las clases que componen el paquete *Camaras*.

Camara

Esta clase crea y configura la cámara que se usará a lo largo del videojuego. En este caso, y para facilitar el manejo de la misma, se optó por usar una cámara que a la vez fuese un nodo. La cámara se comporta como un objeto más dentro del videojuego, como podría comportarse una de las habitaciones o una esfera. Esto facilita en gran medida las operaciones de rotar y mover la cámara, porque no es necesario preocuparse de actualizar los vectores que rigen el comportamiento de la cámara; JMonkey se encarga de ello internamente. La clase que crea cámaras que son nodos a la vez se llama *CameraNode* y se encuentra dentro de JMonkey. Nuestra clase cámara tiene como atributo un objeto de la clase *CameraNode*.

Se decidió usar este tipo de cámaras porque es especialmente útil cuando la cámara debe seguir a un objeto constantemente, girando y desplazándose con él. Al ser la cámara un nodo también, nos permite asociarlo al modelo 3D

del avatar y no tener que comprobar en alguna otra parte del código cuál es la posición del avatar y la de la cámara y tener que actualizarla constantemente.

Entre sus métodos se encuentra métodos para ligar la cámara a otro nodo, rotar y mover la cámara, y métodos accesores para la posición global de la cámara y para el objeto *CameraNode*.

MouseMovingCamPlayer

Esta clase se encarga de gestionar el movimiento de la cámara con el ratón, cuando está ligada al avatar. Al extender la clase *MouseInputAction* de JMonkey, se debe implementar el método *performAction* de esta clase. En este método se realizan dos tareas: en primer lugar permite rotar la cámara en el eje X y en el eje Y dentro de unos rangos, y por otro lado permite variar la posición de la cámara en el eje Y, dentro de unos rangos también.

Todos los parámetros de configuración de los rangos en los que se puede mover la cámara con esta acción de ratón se pasan por parámetro en la constructora. Además utiliza el controlador para saber en qué momento tiene efecto esta acción o no.

MouseMovingCamXZ

Esta clase se encarga de gestionar el movimiento de la cámara con el ratón cuando se va a mostrar la vista aérea. Con esta acción se puede acercar la cámara a los bordes del área visible del videojuego y la cámara se desplazará en la dirección de ese borde. Para conseguir esto se compara la posición actual del ratón con la posición de los bordes. Si la posición coincide con algún borde se varía la posición de la cámara.

La constructora tiene como parámetros el ancho y largo de la pantalla, la velocidad a la que se moverá la cámara, y dentro de qué límites se puede mover la cámara.

MouseMovingTableCamXZ

Esta clase extiende a la clase anterior. Esto es debido a que la vista ofrecida cuando el avatar se acerca a una mesa es muy similar a la ofrecida en la vista aérea del laberinto, pero con una diferencia. El objetivo de la cámara, en este caso, varía en función de qué mesa se acerque el avatar. Por ello se construyó esta clase que funciona de igual manera que la anterior, pero cada vez que se va a comprobar el movimiento de la cámara, primero

se cambia el objetivo de la cámara por la última mesa a la que el avatar del usuario se ha acercado.

4.5.6. Estructura de la GUI

A continuación se explicará el paquete *GUI*, que es el encargado de introducir la parte gráfica con la que interactuará el usuario; es decir ventanas, menus, botones, etc. La figura 4.10 muestra el diagrama UML de estas clases.

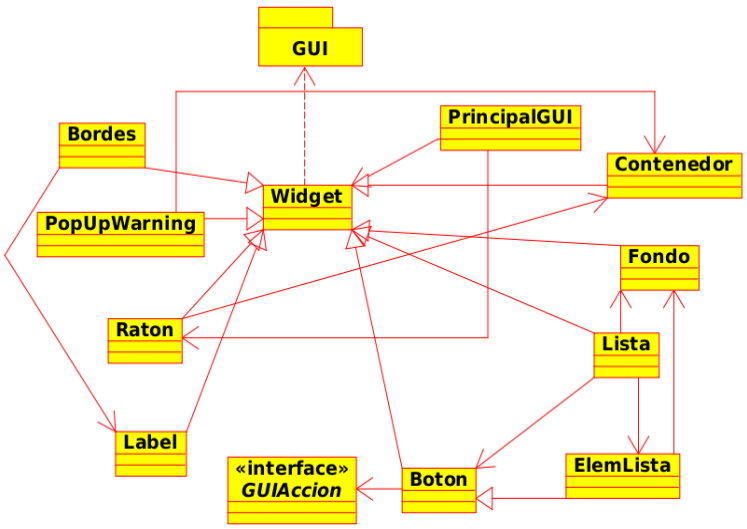

Figura 4.10: Diagrama UML de las clases que componen el paquete *GUI*.

La introducción de la GUI permite que el videojuego adquiera un estilo personal, el cual puede ser cambiado de manera muy sencilla variando solamente las texturas y los colores de cada uno de los componentes. Seguidamente se hablará de manera breve de los componentes más importantes utilizados en la GUI del juego. La explicación se dará en base a las clases del paquete GUI del proyecto para dar una mayor claridad y una mejor visión general al programador:

Componente Bordes

Esta clase se encarga de bordear las diferentes ventanas que aparezcan dentro del videojuego. Para ello se hallan los puntos de los vértices donde se

van a insertar la textura correspondiente. Con ello se pretende dar un estilo personalizado al videojuego y presentar una interfaz más elegante al usuario. Cabe destacar la existencia de métodos para poder insertar y actualizar títulos en dichos bordes.

Componente Botón

Dicha clase se encarga de simular un botón. En su construcción se puede configurar el texto para mostrar en dicho botón, el color de fondo del botón, color de fondo del texto y su tamaño, insertar animación al botón en forma de movimiento y cambiar el aspecto al pasar el ratón por encima de él. Por último, cabe destacar la posibilidad de indicar acciones a realizar cuando se pulse dicho botón. Esta propiedad la hereda del componente Widget, del que se hablará más adelante.

Componente Contenedor

Esta clase se compone de un Widget (se hablará más adelante de él) que sirve para agrupar otros componentes y poder crear una jerarquía. De esta manera es más fácil poder trabajar con los diferentes componentes. Es útil para organizar escenas en las que existen muchos componentes, ya que se insertan en un contenedor y de esta manera sólo hay que configurar el tamaño y la posición de dicho contenedor, obviando los parámetros de cada componente, puesto que anteriormente se han fijado al contenedor.

Componente ElemLista

Clase que se encarga de crear un componente que forma parte del componente lista. En elemLista se prepara la celda para ser insertada en una lista determinada. Aquí es donde se configura el estilo que presentará la celda cuando se seleccione, ya sea mediante ratón o teclado.

Componente Fondo

Esta clase es la encargada de poner fondo a cualquier componente, ya sea listas, contenedores, pop-up, etc. Para ello va fijando los diferentes vértices para luego poder introducir correctamente la textura. Cabe destacar que cambiar el fondo de cualquier componente es tan sencillo como cargar otra imagen en la ruta de dicha textura.

Componente GUIAccion

Clase abstracta que se encarga de declarar acciones que pueden ser ejecutadas por cualquier Widget, es decir en cualquier componente. Dicha clase contiene un método llamado *Accion* que se sobreescribirá para dar las instrucciones correspondientes en cada componente.

Componente Label

Esta clase es la encargada de crear una etiqueta en la GUI. Dicha etiqueta contiene métodos para poder configurar el tamaño del texto, color y poder actualizar el contenido del texto.

Componente Lista

Clase que se encarga de crear una lista. Para ello se tienen que crear los elemLista mencionados anteriormente para ir añadiéndolos a la lista. Lista implementa métodos para poder ajustar el tamaño de la lista, eliminar los botones de scroll, mover el seleccionado, ver qué celda esta seleccionada en ese momento, cambiar la celda seleccionada, etc.

Componente PopUpWarning

Clase que crea un pop-up con información. Se ha de tener cuidado en el orden en el que se genera para que salga visible en primer plano y se sobreponga a los demás componentes que estén en ese momento activos y visibles. El pop-up consta de una pequeña ventana que lleva el borde para no perder el estilo de la GUI y una etiqueta donde se inserta la información que se mostrará. También posee un método que permite activar y desactivar la visibilidad del pop-up, de tal manera que siempre se crea en una escena y lo único que hay que hacer es ponerlo visible en el orden adecuado, como se ha mencionado anteriormente.

Componente Ratón

Esta clase se encarga de dibujar el cursor del ratón en la GUI. Se implementan métodos para poder cambiar el icono por si no se quiere el ratón definido por defecto y también para ocultarlo en caso de que no haga falta en alguna escena determinada.

Componente Widget

Esta clase constituye el componente más importante de la GUI, ya que todos heredan de él. Este componente es el que se encarga principalmente de las acciones y de cambiar el aspecto del componente que herede de él. Este componente implementa métodos para las propiedades : tamaño, visibilidad, activación y la opción de saber si se ha seleccionado o no dicho componente. Cabe destacar que es el componente más relacionado con la arquitectura de JMonkey, ya que es éste el que se encarga de asociar a, y desasociar, de la escena los componentes que heredan de él.

PrincipalGUI

Por último hay que destacar la clase PrincipalGUI. Esta clase es una especie de factoría de Widgets. En ella se inicializan todos los componentes que componen la GUI. Con ello se consigue una mayor claridad y una mejor organización de la GUI. En esta clase es donde se configura la fuente de letra que se utilizará durante la GUI. También hay que destacar que la constructora da la posibilidad con un booleano de que aparezca o no el ratón. Esto se hace de esta manera debido a que existen escenas en el juego que no precisan de ratón, y al iniciar la GUI para esa escena se desactiva. En cada escena del juego se crea un PrincipalGUI para poder utilizar todos los componentes mencionados anteriormente.

4.5.7. Las entradas de usuario

Cada tecla que el usuario vaya a usar para interaccionar con el videojuego debe estar previamente registrada en el manejador de entrada, junto con la acción asociada a dicha tecla. Con el fin de aunar todas estas acciones se han guardado todas en el paquete *Inputs*, y para ayudar a la comprensión de cuál es la utilidad de cada acción, se ha creado la clase *ConfigInput*, que contiene métodos para que, según el estado en el que se encuentre el videojuego, cargar las acciones que sean necesarias. Todas las acciones extienden la clase *InputAction* de JMonkey.

Todas las acciones que aparecen en el videojuego están gestionadas por el controlador, que indica mediante una variable de estado cuándo se puede llevar acabo la acción o no. A continuación se explicarán con brevemente cada una de las acciones.

La figura 4.11 muestra el diagrama UML que explica la organización de las acciones con la clase *Controlador* que las maneja.

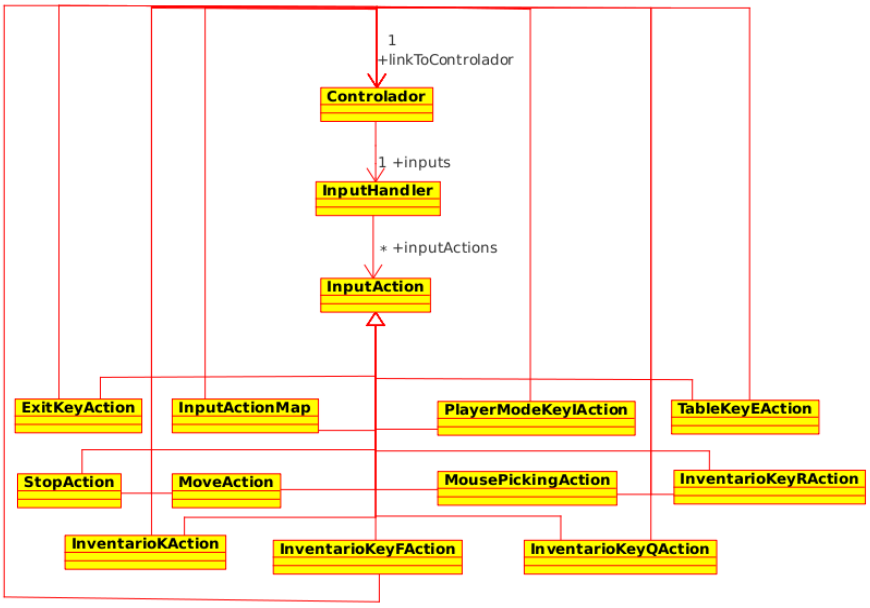

Figura 4.11: Diagrama UML de las clases que compone el paquete *Inputs*.

- *ExitKeyAction* hace que el videojuego termine y la aplicación se cierre. Está asociada a la tecla escape.
- *InputActionMap* hace que la vista cambie para mostrar la vista aérea del laberinto. Está asociada a la tecla **M**.
- *InventarioKeyFAction* y *InventarioKeyRAction* son las acciones que permiten navegar por la lista de objetos del inventario, siempre y cuando éste esté abierto. Están asociadas a las teclas **F** y **R**, respectivamente.
- *InventarioKeyQAction* realiza la acción de preparar un atributo para depositarlo en la mesa en la que el avatar esté actualmente; es decir, guarda el atributo seleccionado en el inventario, cierra el inventario y muestra el puntero del ratón. Está asociada con la tecla **Q**.

- *MousePickingAction* se encarga de buscar qué atributo ha sido seleccionado con el ratón. Una vez lo ha encontrado, en función de la tecla del ratón pulsada y del estado del videojuego se puede: hacer una copia del atributo seleccionado e incluirla en el inventario del avatar, depositar un atributo, previamente seleccionado en el inventario, sobre otro para resolver una dependencia, o abrir un pop-up con información sobre el atributo seleccionado. Esta acción está asociada a las teclas del ratón.

- *MoveAction* se encarga de gestionar el movimiento del avatar. Esta acción está asociada a las cuatro teclas de movimiento (**W, S, D** y **A**). La acción es la misma para las cuatro teclas. Esto se hace así, para poder realizar giros y desplazamientos al mismo tiempo.

- *StopAction* también se encarga de gestionar el movimiento del avatar. Esta acción asociada a las teclas de avance y retroceso (**W** y **S**, respectivamente). Se encarga de detener el avance del avatar cuando estas teclas dejan de presionarse.

- *PlayerModeKeyIAction* se encarga de abrir y cerrar el inventario del avatar. Esto se consigue activando y desactivando el estado de JMonkey (*GameState*) que contiene la GUI del inventario. Esta acción está asociada a la tecla **I**.

- *TableKeyEAction* se encarga de cerrar la vista aérea del laberinto o de salir de la vista centrada en la mesa. Esta acción está asociada a la tecla **E**.

Todas estas acciones se cargan al iniciar un ejercicio y es el controlador el que se encarga de gestionarlas mediante estados y flags de activación. Otra opción podría haber sido ir ligándolas y desligándolas del manejador de entradas de usuario (*InputHandler*) según el estado en el que se encontrase la partida. Esto originaba una gran ralentización del videojuego, ya que el coste de añadirlos y borrarlos es muy elevado.

4.5.8. El controlador

Con el fin de gestionar todos los recursos proporcionados por el motor de videojuegos usado, se creó la clase *Controlador*. Esta clase auna todos los

datos necesarios para gestionar la lógica interna del videojuego, así como para provocar las transiciones entre los distintos estados de JMonkey que se han programado. Entre sus datos se encuentran, por ejemplo, enlaces al avatar del videojuego, un enlace al último atributo seleccionado, con la tecla **Q**, del inventario, un enlace a la última mesa a la que se acercó el avatar, la cámara del videojuego, la lista de atributos que le falta al usuario por sintetizar, enlaces al terreno, al nodo raíz del árbol de objetos 3D de la escena, etc. Además el controlador cuenta con dos atributos enteros que indican el estado en el que se encuentra el videojuego y el inventario. Los posibles estados están descritos mediante constantes. Así, para el atributo que describe el estado del videojuego tenemos los siguientes estados:

- *JUGADOR* nos indica que el usuario puede manejar al avatar dentro del laberinto. Representa el curso normal del videojuego.
- *MESA* nos indica que el usuario ha acercado el avatar a una mesa, lo que producirá un cambio en la cámara y en el comportamiento de algunas entradas. El usuario puede, en éste estado, incorporar nuevos atributos a su inventario o preparar el inventario para depositar un atributo sobre otro que pertenezca a esa mesa.
- *MAPA* nos indica que el usuario está observando el laberinto desde la vista aérea.
- *DEJAR_OBJ* nos indica que el usuario ha seleccionado un objeto de su inventario con la tecla **Q** cuando estaba dentro del estado *MESA* y que se dispone a seleccionar un atributo de la mesa para resolver una dependencia.

Y para el estado del inventario tenemos las constantes que nos indican que está abierto o cerrado. Estos estados se usan en cada una de las acciones de entrada del usuario para determinar cuándo se pueden utilizar dichas acciones u originar comportamientos dependientes de estos estados para determinadas acciones. La idea general es que el controlador está presente en la generación de los elementos que configuran el videojuego para almacenar enlaces a estos objetos para, posteriormente, en las acciones de entrada de usuario poder modificarlos según el estado que nos indique el controlador en cada momento. La figura 4.12 muestra el diagrama UML de la clase *Controlador*.

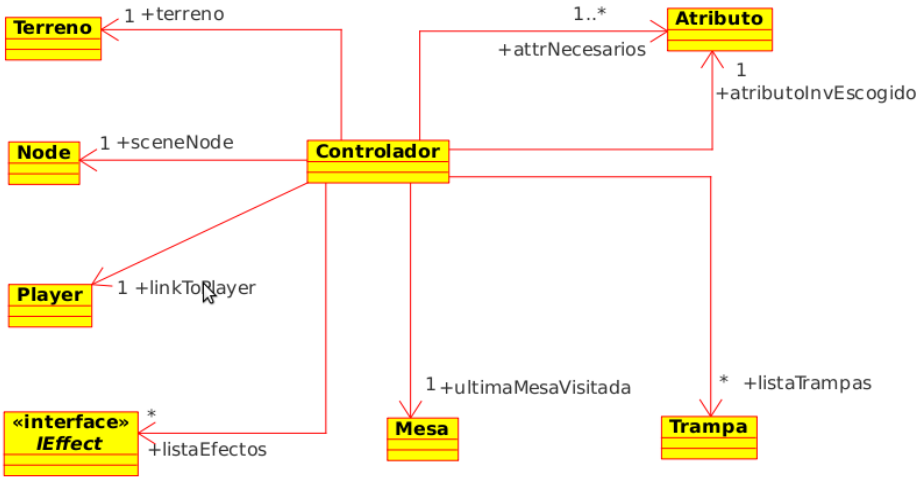

Figura 4.12: Diagrama UML de la clase *Controlador* y su relación con el resto de clases.

4.6. El editor de ejercicios

El editor de ejercicios conforma una aplicación separada del proyecto principal. La aplicación ha sido desarrollada en el lenguaje Java con el toolkit de controles genéricos `SWING`.

El diseño principal de la aplicación es un patrón Modelo/Vista/Controlador.

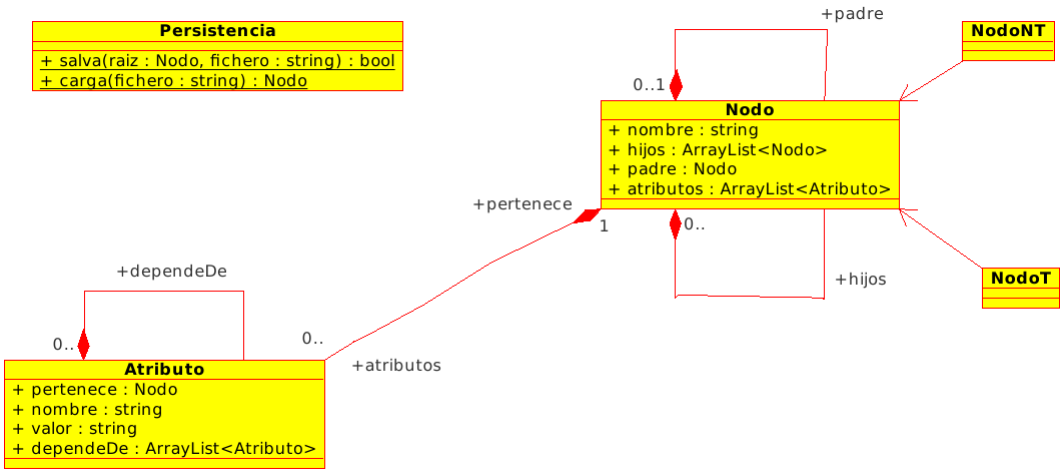

Figura 4.13: Esquema UML del editor de ejercicios.

4.6.1. Modelo

Se ha aprovechado el diseño del proyecto principal, ya que consta de un modelo para generar el mapa en 3D dinámicamente. Este modelo ha sido diseñado de tal forma que aprovecha la tecnología `Serializable`[1] de Java, de tal forma que es posible directamente guardar el estado de determinadas instancias de ciertas clases en ejecución a un fichero. En este caso, se ha optado por el formato XML.

Una vez volcado el estado del modelo a un fichero XML, se podrá cargar dicho fichero XML en el juego Knuthians, siendo instanciado este modelo, ya que ambos modelos, tanto el del juego como el del editor de ejercicios coinciden.

4.6.2. Vista

La vista es un control `JTree`[2] del toolkit `SWING` de Java. Se ha aprovechado el modelo escrito para serializar el estado del mapa en un fichero para mostrar este árbol al usuario y modificarlo en tiempo de ejecución para diseñar nuevos mapas y retos.

[1] http://java.sun.com/javase/6/docs/api/java/io/Serializable.html
[2] http://java.sun.com/javase/6/docs/api/javax/swing/JTree.html

4.6.3. Posibles mejoras

El editor de ejercicios podría ser ampliado en un futuro para añadir un paso más en el que se puedan determinar las posiciones de las trampas en los pasillos y habitaciones, de tal forma que estas trampas puedan ser establecidas a priori por el tutor que diseñe el mapa o el reto.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

En este trabajo se ha desarrollado un videojuego educativo que pretende enseñar el funcionamiento de las gramáticas de atributos. También se ha creado una herramienta orientada a la creación de ejercicios para el videojuego. Se ha mostrado, también, comó se usa el videojuego y la herramienta de creación de ejercicios. De esta forma, el trabajo ha satisfecho completamente los objetivos planteados al comienzo del mismo. Más concretamente:

- El trabajo nos ha permitido iniciarnos y profundizar en el conocimiento de las tecnologías para el desarrollo de videojuegos, lo que ha resultado un valioso complemento a los conocimientos adquiridos en nuestros estudios de Ingeniería Informática. Así mismo, también hemos profundizado en los conocimientos propios de la materia de Procesadores de Lenguaje.

- Nos ha permitido introducirnos en el mundo del e-Learning. Más concretamente, nos ha introducido a la creación de videojuegos educativos, sus usos y sus características.

- Nos ha ayudado a mejorar nuestras habilidades a la hora de trabajar en grupo y de forma colaborativa en un proyecto, con lo que hemos profundizado en los conocimientos propios de la materia de Ingeniería del Software.

5.2. Trabajo futuro

El trabajo iniciado en este proyecto no termina, en absoluto, aquí. Efectivamente, surgen múltiples líneas de trabajo futuro. Estas líneas pueden clasificarse en líneas a corto, medio y largo plazo:

- A corto plazo surgen distintas mejoras para dar al videojuego un mejor aspecto y hacerlo más atractivo. Por un lado, es necesario mejorar el apartado gráfico del videojuego, creando modelos 3D más elaborados para el avatar y el resto de elementos del videojuego. Así mismo, sería conveniente mejorar las animaciones del avatar. Por otro lado, es necesario introducir algunos elementos lúdicos adicionales al videojuego para hacerlo más atractivo. Algunas ideas para conseguirlo pueden ser: añadir enemigos con los que el jugador deberá lidiar o crear trampas más elaboradas. También, con el fin de aumentar la componente educativa, se podría añadir alguna trampa que obligase al usuario a responder a una pregunta relacionada con la asignatura de Procesadores de Lenguajes. Por último, sería necesario realizar una refactorización y mejora de la herramienta de creación de ejercicios, para hacer su uso más cómodo e intuitivo.

- A medio plazo surge la necesidad de investigar la utilidad educativa de este videojuego. Para ello se propone realizar estudios entre el alumnado de la asignatura de Procesadores de Lenguajes, mediante sencillos test para comprobar la utilidad de forma estadística. En función de este estudio se realizarían los cambios necesarios en él.

- A largo plazo surgen distintas mejoras para completar el producto. Entre ellas, construir un sitio web donde promocionar el videojuego, con tutoriales sobre su uso, una comunidad web donde los usuarios puedan compartir sus experiencias y consejos, etc. Otra posible mejora es la de dotar de un sistema automático de descargas de ejercicios al videojuego. Con este sistema el docente podría subir sus ejercicios a un servidor web y el alumno, bajarlos automáticamente desde el servidor.

Capítulo 6

Referencias

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (second edition). Addison-Wesley. 2007.

2. Paakki, J. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation, ACM Computer Surveys, vol. 27, no. 2, pp. 196-255, 1995.

3. D.E. Knuth. Semantics of Context-free Languages, Math.System Theory 2(2), 127-145, 1968.

4. D. E. Knuth. Semantics of Context-free languages: Correction. Math. Systems Theory, 5(1): 95-96, 1971.

5. Garcia Peñalvo Francisco José . Estado actual de los sistemas e-Learning. Revista Teoría de la Educación Nº6. 2005-2006.

6. Nicholson Paul. A History of e-Learning. Computer and Education. Springer. 2007.

7. Gómez Martín Marco Antonio. Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos. Tesis doctoral. 2007.

8. Iván Martínez-Ortiz, Pablo Moreno-Gervas, Jose Luis Sierra, Baltasar Fernández-Manjón. Production and Deployment of Educational Videogames as Assessable Learning Objects. 2006.

9. Garris, R., Ahlers, R., and Driskell, J.E., Games, Motivation and Learning: A Research and Practice Model. Simulation and Gaming. Vol 33(4). 2002. 441-467.

10. Stephen Downes. Learning Objects: Resources for distance education worldwide. The International Review of Research in Open and Distance Learning, Vol 2, No 1 (2001), ISSN: 1492-3831.

11. Daniel Burgos, Pablo Moreno-Ger, Jose Luis Sierra, Baltasar Fernandez-Manjon, Rob Koper. Authoring game-based adaptive units of learning with IMS Learning Design and e-Adventure. International Journal of Learning Technology Volume 3, Number 3 / 2007.

12. Esther del Moral. "Juegos de rol, aventuras gráficas y videojuegos: la creatividad lúdica a través del software". Aula de Innovación educativa, 50, p. 63-67. 1996.

13. Landa Cosio Nicol. DirectX Programación de gráficos 3D. MP Ediciones 2007.

14. Francis S. Hill, Jr. and Stephen M. Computer Graphics Using Open GL. Kelley. 2007.

15. Derakhsani, Dariush. Maya 2008 (diseño y creatividad). Anaya Multimedia. 2008

16. 3DSMAX 2010 El gran libro. VV.AA. Marcombo. 2010.

17. Koenigsmarck, Arndt Von. Cinema 4D 10. Anaya Multimedia 2007.

18. Mullen, Tony. Animación de personajes con Blender (diseño y creatividad). Anaya Multimedia. 2007.

19. Jenkins López, David B. y Monte Freitas, Álvaro del y Montenegro Montes, Manuel. Rigid Body Simulation. 2006.

    http://eprints.ucm.es/9043/

20. Bradley Ford. QuickTime Pro 5y6. Ed. Anaya Multimedia. 2003.

21. GoodMan, Danny. JavaScript y dhtml. Anaya Multimedia. 2008.

22. Liberty, Jesse. Programming C# 3.0 (5 ED). O'Reilly and Associates. 2008.

23. Sharp, John. Microsoft Visual C 2010 step by step. 2010.

24. Wall, Kurt. The definitive guide to GCC. Apress.

25. James Bucanek. Beginning Xcode (Programmer to Programmer). 2006.

26. Enrique Hernández Orallo, José Hernández Orallo y Mª Carmen Juan Lizandra. C++ estándar. Paraninfo Thomson Learning. 2001.

27. Rémi Arnaud y Mark Barnes. Collada sailing the gulf of 3D digital content creation. 2006.

28. Charles River Media. The OpenAL Programming Guide (Charles River Media Game Development). 2006.

29. A. Fernández-Valmayor, A. Navarro, B. Fernández-Manjón y J. L. Sierra. «Lenguajes de programación, lenguajes de marcado y modelos hipermedia: una visión interesada de la evolución de los lenguajes informáticos». Madrid: Universidad Complutense. 2006.

30. David Hunter...[et al]. Beginning XML. Indianapolis, IN : Wiley Publishing, cop. 2007.

31. Charles F.Goldfarb. The SGML handbook. Oxford : Clarendon Press. 1998.

32. Eric van der Vlist. XML Schema. Beijing ; Sebastopol, CA : O'Reilly. 2002.

33. Doug Tidwell. XSLT. Sebastopol, Calif. : O'Reilly. 2008.

34. Tejeiro Salguero, Ricardo y Pelegrina Del Rio, Manuel. Los videojuegos: Qué son y como nos afectan. Ariel. 2003.

35. Hearn D. D. y Baker M. P. Gráficos por Computadora. Prentice Hall, Pearson, 2005, 3.a edición.

36. Stephen Cawook, Mark Fiala. Augmented reality: a practical guide. The Pragmatic Bookshelf. 2008.